Archiving Scientific Data

PETER BUNEMAN
University of Edinburgh
SANJEEV KHANNA
University of Pennsylvania
KEISHI TAJIMA
Japan Advanced Institute of Science and Technology
and
WANG-CHIEW TAN
University of California, Santa Cruz

Archiving is important for scientific data, where it is necessary to record all past versions of a database in order to verify findings based upon a specific version. Much scientific data is held in a hierachical format and has a key structure that provides a canonical identification for each element of the hierarchy. In this paper we exploit these properties to devlop an archiving technique that is both efficient in its use of space and preserves the continuity of elements through versions of the database, something that is not provided by traditional minimum-edit-distance diff approaches. The approach also uses timestamps. All versions of the data are merged into one hierarchy where an element appearing in multiple versions is stored only once along with a timestamp. By identifying the semantic continuity of elements and merging them into one data structure, our technique is capable of providing meaningful change descriptions, the archive allows us to easily answer certain temporal queries such as retrieval of any specific version from the archive and finding the history of an element. This is in contrast with approaches that store a sequence of deltas where such operations may require undoing a large number of changes or significant reasoning with the deltas.

A suite of experiments also demonstrates that our archive does not incur any significant space overhead when contrasted with diff approaches. Another useful property of our approach is that we use XML format to represent hierarchical data and the resulting archive is also in XML. Hence, XML tools can be directly applied on our archive. In particular, we apply an XML compressor on our archive, and our experiments show that our compressed archive outperforms compressed diff-based repositories in space efficiency. We also show how we can extend our archiving tool to an external memory archiver for higher scalability and describe various index structures that can further improve the efficiency of some temporal queries on our archive.

Categories and Subject Descriptors: H.2.8 [**Database Management**]: Scientific Databases; H.3.7 [**Information Storage and Retrieval**]: Collection, Dissemination, Systems issues; H.2.1 [**Database Management**]: Data models

General Terms: Algorithms, Design, Experimentatal Analysis, Management

Additional Key Words and Phrases: Keys for XML

Authors' addresses: Peter Buneman, Email: peter@cis.upenn.edu; Sanjeev Khanna, Email: sanjeev@cis.upenn.edu; Keishi Tajima, Email: tajima@jaist.ac.jp; Wang-Chiew Tan, Email: wctan@cs.ucsc.edu.

1. INTRODUCTION

Scientific databases are published on the Web to disseminate the latest research. As new results are obtained these databases will be updated, mainly by adding data, but also by modifying and deleting existing data. Since other research may be based on a specific version of a database, it is important to create archives containing all previous states of the data. Failure to do so means that scientific evidence may be lost and the basis of findings cannot be verified. A search of scientific data available on the Web [CELLBIODBS] suggests that archiving is an ubiquitous problem. Even databases of physical constants [PHYSICSCONSTANTS] are less "constant" than one might naively imagine. The onus of keeping archives typically falls on the producers of data, but there are no general techniques for efficiently keeping long-term archives that also provide efficient support for basic operations such as retrieval of a past version from the archive and tracing the evolutionary history of an element in the archive.

As an example of the issues involved, consider two widely used datasets in genetic research: Swiss-Prot [Bairoch and Apweiler 2000], a protein sequence database, and On-line Mendelian Inheritance in Man (OMIM) [OMIM 2000], a database of descriptions of human genes and genetic disorders. Both databases have a similar hierarchical structure and both are heavily *curated*, i.e., they are maintained with extensive manual input from experts in the field. In the case of Swiss-Prot, a new version is produced approximately every four months, and all old versions are kept. On the other hand, in the case of OMIM, a new version is produced almost every day, but only occasionally is a (printed) archive produced. Swiss-Prot and OMIM are two contrasting examples of archiving practices[1]. There is an obvious trade-off between the frequency with which the database is "published" and the space required for complete archiving. Although keeping all old versions allows one to quickly obtain an old version, such an archiving method clearly does not scale well in the long run or if the database is published very frequently, such as like OMIM. Even if the issue of space is not critical, and we choose to keep all versions, there is also the issue of the efficiency with which one can query the temporal history of some part of the database. For example, to find when a given observation first appeared in history or when it was last changed may require one to locate that observation in each of a very large number of versions.

Another approach to archiving is to keep a record of the changes – a "delta" – between every pair of consecutive versions. We will call this the *sequence-of-delta approach* throughout the paper. Such a method of archiving clearly conserves space and scales well. In this approach, however, retrieving an old version might involve undoing or applying many deltas. Likewise, the efficiency of finding the evolutionary history of an element is also a problem and may require significant

[1]It should be mentioned that, in addition to the published versions, both SWISS-PROT and OMIM keep audit trails of the edits to the data, so there is more historical information than is apparent from the archives. However it is not clear that either organization could easily produce the state of its data as it was at some arbitrary past time.

Version 1

```
<gene>
   <id>6230</id>
   <name>GRTM</name>
   <seq>GTCG...</seq>
   <pos>11A52</pos>
</gene>
<gene>
   <id>2953</id>
   <name>ACV2</name>
   <seq>AGTT...</seq>
   <pos>08A96</pos>
</gene>
```

Version 2

```
<gene>
   <id>2953</id>
   <name>ACV2</name>
   <seq>GTCG...</seq>
   <pos>11A52</pos>
</gene>
<gene>
   <id>6230</id>
   <name>GRTM</name>
   <seq>AGTT...</seq>
   <pos>08A96</pos>
</gene>
```

Output of diff

```
2,3c
   <id>2953</id>
   <name>ACV2</name>
8,9c
   <id>6230</id>
   <name>GRTM</name>
```

Fig. 1. Two versions and the diff.

reasoning with the deltas.

Tools for keeping changes made to text documents, such as CVS [CVS] often adopt the sequence-of-delta approach. Such tools typically use line-diff algorithms [Miller and Myers 1985; Myers 1986] to describe changes. Tree diffs have also been developed for XML and other hierarchical structures [Zhang and Shasha 1989; Chawathe et al. 1996; Chawathe and Garcia-Molina 1997; Cobena et al. 2001]. These diff algorithms compute deltas that are also based on the notion of minimal-edit-distance, and we shall refer to these as *diff-based aproaches*. Measuring the "diff" of two structures is one of a number of ways of computing a delta, others could be based, for example, on edit scripts or transaction logs. The problem with the diff-based approach is that it may ignore the semantic continuity of data elements. As an example, suppose we have a database containing data of two genes, as shown in Version 1 of Figure 1. It was later discovered that the information on one gene had been confused with the other, and the data was corrected as shown in Version 2. A diff algorithm might explain the change as genes changing their names and id numbers, as shown by the diff output in Figure 1; It says that lines 2-3 of Version 1 should be replaced with `<id>2953</id><name>ACV2</name>`, i.e., the gene GRTM changed its id to 2953, and also changed its name to ACV2. Similarly, the diff says the gene ACV2 changed its id to 6230, and also changed its name to GRTM.

If we are only interested in retrieving an entire (past) version from a diff-based repository, such nonsensical change descriptions do not surface. However, if we are interested in finding the temporal history of an element in the database, the diff-based approach may require one to perform some complicated analysis on the diff scripts. This example suggests that there is a temporal invariance of keys that should be captured by an archiving system. We would like an archiving system that is able to preserve the continuity of data elements through time.

The archiving techniques described in this paper stem from the requirements and properties of scientific databases. They are, of course, applicable to data in other domains, but they are especially appropriate to scientific data for a variety of reasons. First, as we have noted, archiving is important for scientific data. Second, scientific data is largely accretive or ”write mostly”, and the transaction rate is low. In particular, an individual data entry will be modified infrequently. Third, much scientific data is kept in well-organized hierarchical data formats and is naturally converted into XML. Finally, this hierarchically structured data usually has a *key structure* which is explicit in the design of the format. The key structure provides a canonical identification for every part of the document, and it is this structure that is the basis for our techniques.

Our archiver leverages these properties and effectively stores multiple versions of hierarchical data in a compact archive that can easily support certain temporal queries by using the following techniques:

—**Identifying changes based on keys.** In contrast to the many existing tools that use the diff-based approach based on minimum edit distance, we identify the correspondence and changes between two given versions based on keys. We will call this approach *key-based approach*. By this, our archive can preserve semantic continuity of each data element in the archive, and can efficiently support queries on the history of those elements.

—**Merging versions based on keys.** In contrast to the delta-based approach, we merge all versions into one hierarchy. We will call this approach *merging approach*. An element may appear in many versions, but we identify those occurrences using the key structure and store it only once in the merged hierarchy. The sequence of versions in which an element appears is described by its *timestamp* (a sequence of version numbers) and is stored together with that element. Since changes to our database are largely accretive and an element is likely to exist for a long time, we can compactly represent its timestamp using *time intervals* rather than a sequence of version numbers. Notice that the description of changes is grouped by elements in this structure while the description of changes is grouped by time in the sequence-of-delta approach. When compared with the latter approach, the former can easily support both the retrieval of very old versions or queries on the history of an element.

—**Inheritance of timestamps.** Conceptually, a timestamp is stored with every element to indicate its existence at various points in history. In actual implementation, a timestamp is stored at a child element only when it is different from the timestamp of its parent element. Since most changes are insertions of new elements, it is likely that an element in the archive has identical timestamp to its parent element. Hence a fair amount of space overhead is usually avoided by inheriting timestamps.

There are two orthogonal choices to be made among the possible descigns of an archiver: whether to use a diff-based or key-based method of describing change and whether to use a sequence-of-delta or a merging technique in recording the change. Our approach is key-based + merging approach while systems such as CVS use a diff-based + sequence-of-delta approach. The other two combinations are also possible, and in fact, we also use diff-based + merging approach for unkeyed part of the data. The details are explained later.

Another interesting aspect of our approach is that our archive can be easily represented as yet another XML document. Our choice of using XML is motivated in

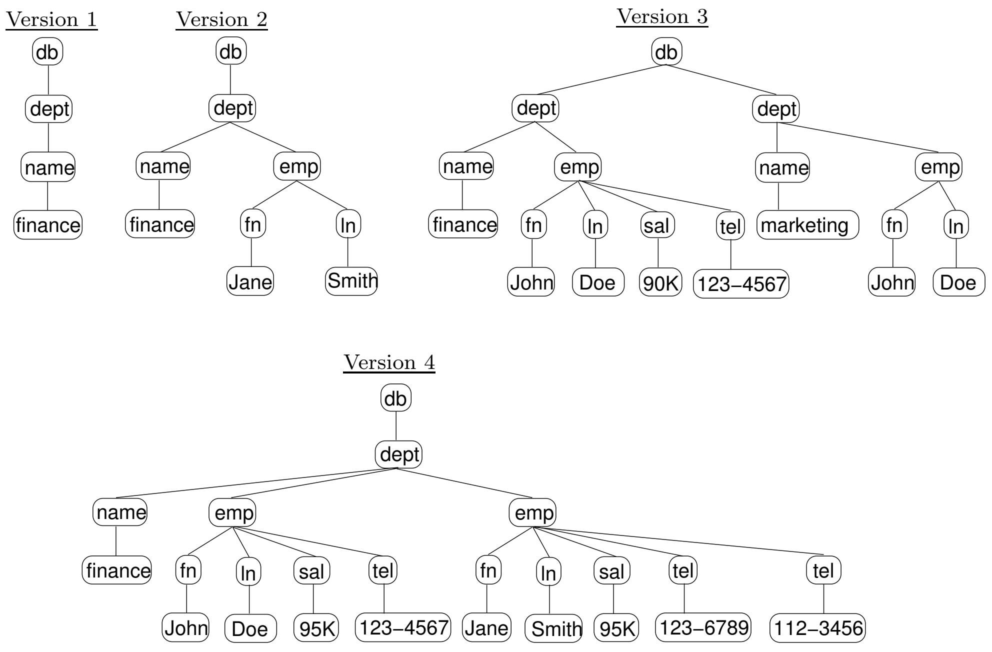


Fig. 2. A sequence of versions

part by the existence of many biological databases in hierarchical formats, which are close to the XML data model. Many of these databases are also moving towards support for XML (e.g., Swiss-Prot [EMBL-EBI (European Bioinformations Institute)]). Since XML is also a widespread format for representation and exchange, we assume sources are provided in XML and chose to represent our archive in XML. Our choice of XML also allows us to leverage many existing XML tools such as XML editors, parsers, etc. that are readily available. In particular, we subsequently use an XML compressor such as XMill [Liefke and Suciu 2000] to further compress the archive (see Section 5).

This paper describes the archiving technique in detail and gives some experimental data on its performance. The technique is an extension of the "persistent data structures" of Driscoll et al. [Driscoll et al. 1989] (see Section 8). In order to establish the viability of this approach, we do several things:

—We show that the compacted archive can be constructed efficiently, i.e., a new version can be efficiently merged with an existing archive.

—We show, experimentally, that the space overhead for our compacted archive is comparable to the existing approaches.

—We also establish experimentally that our compacted archives work well with XMill, and outperform compressed diff-based repositories in terms of space efficiency.

—Since our archive preserves the semantic continuity of elements, it is easy to support various temporal queries more efficiently than delta-based approaches. We also show that we can further improve the efficiency of answering certain

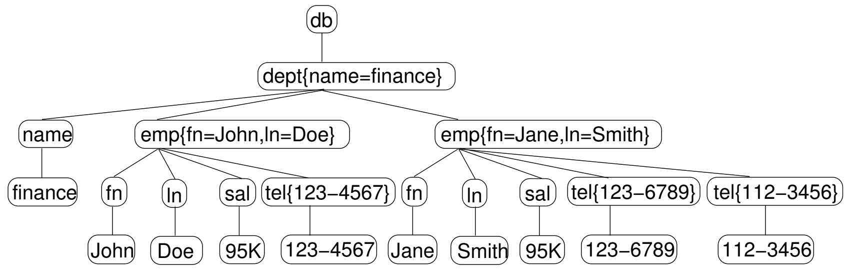


Fig. 3. Version 4 annotated with key values.

queries by defining simple index structures on top of our archive.

For example, on the basis of our experimental findings for OMIM, which is not adequately archived (we recorded 100 versions over roughly 100 days), we predict that we should be able to construct a compacted archive for a year in less than 1.12 times the space of the last version. Moreover, the archive, under XMill, will compress to 40% of the size of the last version.

2. AN EXAMPLE

A simple example illustrates how the archiver works. In Figure 2, we see a sequence of versions of a company database containing information about its employees in each department. Every department can be identified by its department name, i.e., name is the key for departments. Within each department, every employee can be uniquely identified by the employee's firstname and lastname, i.e., firstname and lastname is the key for employees. Notice that it is possible for two employees have the same firstname and lastname if they belong to distinct departments (e.g., John Does of version 3). Each employee also has at most one salary value, and one or more telephone numbers. Figure 3 shows version 4 of Figure 2 with key information annotated on nodes. For example, emp nodes are annotated with firstname and lastname values and tel nodes are annotated with its subvalue.

Figure 4 shows how those versions can be merged into one compact archive by "pushing down" timestamps. We first start at the top level of the versions, and determine the nodes that correspond to one another across all versions according to their key values. At the top level, each version has only one root node and they correspond to one another. We merge corresponding nodes together, annotate the resulting node with the version numbers of all merged nodes and push the timestamp of each merged node down their respective subtrees. We recursively invoke this procedure for the children of merged nodes until we reach the leaves. The root node in the archive is used to keep track of the possibility that an archived version is empty. For example, suppose the database is empty at version 5. If version 5 is archived, then the node root will have timestamp t=[1-5] while the node db will have timestamp t=[1-4], indicating that version 5 is an empty database[2].

[2]Note that the current version number is the largest number in the root node. Instead of using a root node, one can also choose to keep track of the next version number as meta-information and increment the next version number whenever a version is archived.

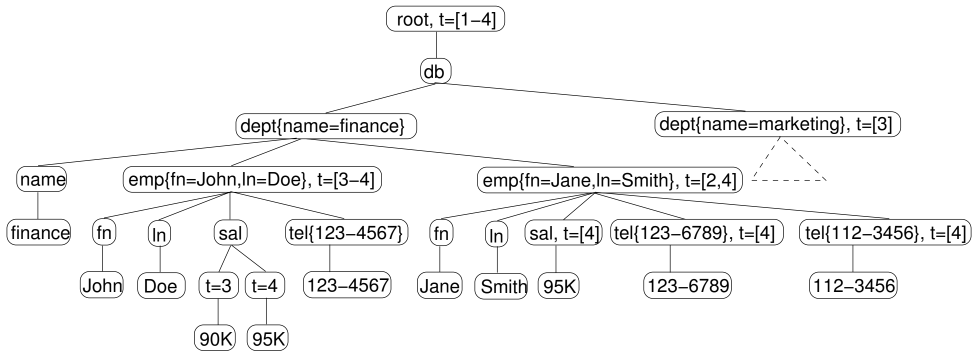


Fig. 4. "Pushing" time down. Example of an archive.

Observe that in the resulting archive in Figure 4, nodes appearing in many versions are stored only once in the archive. If a node occurs in version i, then the timestamp of the corresponding node in the archive contain i. We use time intervals to describe the sequence of versions for which a node exists. For example, the time intervals [1-3,5,7-9] denotes the set {1,2,3,5,7,8,9}. If a node does not have a timestamp, it is assumed to inherit the timestamp of its parent. For example, the finance dept node inherits the timestamp t=[1-4] from its parent db node. Observe also that it is a property of the archive that the timestamp of a node is always a superset of timestamps of any descendant node. This archive can be represented in XML, as shown in Figure 5. For example, employee John Doe of the finance department has a timestamp `<T t="3-4">` around it, indicating that the entire subtree within exists in versions 3 and 4. Furthermore, during these times, John has salary 90K at version 3 and 95K at version 4.

We may assume that the tag T is in a separate namespace [W3C 1999a]. It is also easy to see that to reconstruct any previous version, all we need is a simple scan through the "compacted" document. Notice that information on changes is grouped by elements in this structure while information on changes is grouped by time in the sequence-of-delta approach. Also, our archive ignores the order among elements with keys. If the emp node corresponding to John Doe occurs in version 5 after the emp node corresponding to Jane Smith, it might still be represented in the archive before Jane Smith.

There are two immediate caveats about this approach. First, what happens if the data does not have a key system, i.e., there are nodes that cannot be uniquely identified by their paths and any subelements? We have found that most scientific data sources have a well-organized key system. However, there are cases in which we cannot define appropriate keys for all the nodes down to the leaves. For example, some data may be free text represented as a sequence of <line> elements and some <line> elements may have same text value. Even in such a case, provided the upper nodes in the tree have keys, we can still "push down" the timestamps in the upper part, stop merging, and apply a conventional diff algorithm when we reach elements without keys, such as <line> elements in the example above. In fact, if the entire document does not have appropriate keys, our archiving technique is very close to the SCCS approach [Rochkind 1975] (see also Section 8). The second caveat is

```
<T t="1-4">
   <root>
      <db>
         <T t="3">
            <dept>
               <name>marketing</name>
               <emp>
                  <fn>John</fn> <ln>Doe</fn>
               </emp>
            </dept>
         </T>
         <dept>
            <name>finance</name>
            <T t="3-4">
               <emp>
                  <fn>John</fn> <ln>Doe</ln>
                  <T t="3"><sal>90K</sal></T>
                  <T t="4"><sal>95K</sal></T>
                  <tel>123-4567</tel>
               </emp>
            </T>
            <T t="2,4">
               <emp>
                  <fn>Jane</fn> <ln>Smith</ln>
                  <T t="4"><sal>90K</sal></T>
                  <T t="4"><tel>123-4567</tel></T>
                  <T t="4"><tel>112-3456</tel></T>
               </emp>
            </T>
         </dept>
      </db>
   </root>
</T>
```

Fig. 5. XML representation of the archive in Figure 4.

whether the new structure really is smaller than the accumulated past versions. Here, we capitalize on the fact that the timestamps associated with elements can be compactly represented as a small number of intervals and are often inherited from a parent element. Our experiments in Section 5 validate this.

In the following sections we first explain the notion of keys for hierarchical data that plays an important role in our archiving system. Next, we describe the main modules of the archiver in detail, including key annotation, nested merge and fingerprinting of keys for efficiency. We then show some experimental results on some widely used biological data sets and also on some synthetic data. The following sections show how the whole process can be implemented on large data sets using external memory and show how versions and the temporal history of an element may be efficiently retrieved.

3. KEYS FOR HIERARCHICAL DATA

The general notion of a key for XML is described in [Buneman et al. 2001] and summarized in Appendix A. Informally, a key is an assertion of the form $(Q, \{P_1, \ldots, P_k\})$ where $Q, P_1, \ldots, P_k$ are all path expressions in a syntax like XPath [W3C 1999b] but consisting only of node and attribute names. The path Q identifies the *target set* for the key – the set of nodes to be constrained by the key. The paths $P_1, \ldots, P_k$ are the *key paths* and are analogous to the key attributes in the relational setting. An XML document *satisfies* a key $(Q, \{P_1, \ldots, P_k\})$ if

—from any node identified by Q, every path P_i $(1 \leq i \leq k)$ exists uniquely, and

—if two nodes n_1 and n_2 identified by Q have the same value at the end of each key path in $\{P_1, \ldots, P_k\}$, then n_1 and n_2 are identical nodes.

Example: (/book, {isbn}). Every book child of the root must have a unique isbn child, and if two such book nodes have the same value for their isbn children, then they are the same node.

In the definition above, the path Q in a key starts at the root. We shall need to be able to define keys starting from one of a set of "context nodes". For example we would like to say that beneath any book node, a chapter node is specified by its chapter-number child. But this is to hold only beneath some book node – not globally. We can specify such a key simply by adding to the definition the context path, in this case (/book, (chapter, {chapter-number})). Such a key is called a *relative key*. All the keys we deal with will be relative.

As a further example, version of the company database satisfies the following key constraints.

—(/, (db, {})). There is at most one db element below the root.

—(/db, (dept, {name})). Every dept node within a db node can be uniquely identified by the contents of its name subelement.

—(/db/dept, (emp, {fn, ln})). Every emp node within a dept node along the path /db/dept can be uniquely identified by the contents of its fn and ln subelements.

—(/db/dept/emp, (sal, {})). There is at most one sal subelement under each emp node along the path /db/dept/emp.

—(/db/dept/emp, (tel, {.})). Every tel node within an emp node along the path /db/dept/emp can be uniquely identified by its contents ("." denotes the empty path). That is, the same telephone number cannot be repeated below an emp node.

The keys shown above are rather simple. For the scientific data that we use for our experiments, keys are also simple. (See Appendix B.) Observe that for strong keys, whenever a key $(Q, (Q', \{P_1, ..., P_k\}))$ exists, the keys $(Q/Q', (P_i, \{\}))$, $1 \leq i \leq k$ are implied (P_i is a non-empty key path). Although we do not explicitly state these keys, we shall always assume that they are part of the key specification. We remark that for documents that are standard and consistent representations of relations in XML, the set of keys can be automatically generated from the relational schema.

Some terminology. We say a node or an element is *keyed* if it has a key. In other words the sequence of tag names from the root to this node is equal to the concatenation of context and target path of some key. In the example databases in Figure 2, every non-leaf node is keyed. Given a set of keys, we consider the paths given by the concatenation of context and target paths for every key in the key specification. We say a path in this set is a *frontier path* if it is not a proper prefix of some other path in the set. A node is a *frontier node* if the sequence of tag names from root to that node equals to some frontier path. In other words, a frontier node is the deepest possible keyed node. For example, the key specification for the company database has frontier paths /db/dept/name, /db/dept/emp/fn, /db/dept/emp/ln, /db/dept/emp/sal, and /db/dept/emp/tel. An example of a frontier node is name, but the node emp is not a frontier node. Obviously, there can be unkeyed nodes beyond frontier nodes. If there are area_code and number subelements within each tel node, then these subelements are unkeyed nodes beyond the frontier node tel.

Assumptions about the key structure. In this paper, we assume the given key specification possess the following properties.

—We assume that every key defined for a node is relative to its parent node and keys are defined up to a certain depth in the tree. For example, to identify an emp node that lies on the path /db/dept/emp requires one to first identify the correct db and dept nodes. That is, the key defined for an emp node is relative to its parent node, dept, and the key defined for a dept node is relative to its parent node, db. Such keys are called "insertion-friendly" keys in [Buneman et al. 2001].

—Given a set of keys that a document satisfies, every node that does not occur beneath frontier nodes is keyed. In other words, we assume that the keys "cover" every node that does not occur beneath frontier nodes. For example, there cannot be some unkeyed node directly under emp since emp nodes are not frontier nodes.

—Our last restriction is that nodes that exist beneath some key path cannot be keyed. Given a key $(Q_2, (Q'_2, \{...\}))$, there cannot be a key $(Q_1, (Q'_1, \{P_1, ..., P_k\}))$ where $k \geq 1$ such that $Q_1/Q'_1/P_i$ is a proper prefix of Q_2/Q'_2, for some $i \in [1, k]$. The reason is that the order among keyed nodes is unimportant and since the order among element nodes in an XML value is important for (key) value equality, the nodes beneath $Q_1/Q'_1/P_i$ for every $i \in [1, k]$ should not be keyed. Otherwise, the key of a node in Q_1/Q'_1 may be different each time the order among keyed nodes beneath $Q_1/Q'_1/P_i$ changes.

The first assumption makes it easy for us to determine the correspondences between nodes in the archive and a version as we go down the trees starting from the root. Although this assumption may seem restrictive, we have found that the keys for our experimental data naturally adhere to this assumption. The second assumption is a restriction made only to simplify the discussion of our merge algorithm in Section 4.2. In the event that non-keyed nodes above frontier nodes exist, our archiver handles these nodes by applying conventional diff techniques on the nodes, since no key information is known about these nodes. The last assumption is in fact not required due to the "top-down" localized merging of nodes in our merge algorithm as discussed in Section 4.2. Two nodes in Q_1/Q'_1 are merged only when their corresponding values at $Q_1/Q'_1/P_i$ are equal. In case the two nodes at Q_1/Q'_1 are merged, the two corresponding nodes at Q_2/Q'_2 are also merged by virtue that they occur in a key path of Q_1/Q'_1.

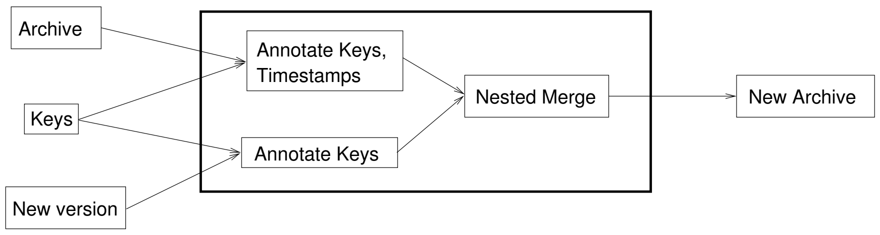


Fig. 6. Main modules in our archiving system.

4. MAIN MODULES

We now describe our main modules: Annotate_Keys and Nested_Merge as shown in Figure 6. The module Annotate_Keys annotates each keyed element in a document or an archive with its key value. For example, Annotate_Keys will transform the version 4 of the database in Figure 2 into the database shown in Figure 3. The key values allow Nested_Merge to determine immediately the corresponding nodes in the archive and version. A node in the archive corresponds to a node in the version if they have the same key value. The module Nested_Merge merges a new version into the existing archive according to correspondence of nodes between the archive and version. When annotating archives, we also need some processing of the timestamps, which will be explained later.

4.1 Annotate Keys

In this section, we show how, given a key specification, one can annotate every keyed node in an XML document with its key value. For example, emp nodes in the version of Figure 3 and the archive of Figure 4 are annotated with these keys — emp{fn=Jane, ln=Smith} and emp{fn=John, ln=Doe}. The resulting tree is such that every keyed node can be identified by the path from root to that node by taking into account existing key values as well.

The idea behind the algorithm Annotate_Keys is to scan through the document, looking out for nodes that should be keyed and nodes that contain key path values. Given a key $(Q, (Q', \{P_1, ..., P_k\}))$, a node that is on the path Q/Q' is a node to be keyed and a node that is on the path $Q/Q'/P_i$, where $i \in [1, k]$, is a node that contains a key path value. (The key path value is the XML value rooted under that node.) As an XML document is scanned in document order, an open tag is always encountered before the corresponding close tag (assuming that the document is well-formed). Alternatively, we say a node is entered first before it is exited in the tree representation of the XML document. Observe that by the time one exits a node n that needs to be keyed (i.e., n is along the path Q/Q'), the nodes containing key path values of n would already have been visited (i.e., those nodes are along paths $Q/Q'/P_i$, $i \in [1, k]$).

Whenever we encounter a node that contains a key path value, we memorize the entire "subtree" of the node and store the subtree value at the node. One can avoid storing large key path values by computing and storing its fingerprint instead. We describe how one can compute fingerprints in Section 4.3. We emphasize that fingerprints are used entirely for efficiency purposes. A fingerprint uses less space than the actual value and it is faster to compare two fingerprints (comparisons are done in Nested_Merge described in Section 4.2) of the respective key values instead of comparing the actual key values. In fact, most well-organized databases have small key path values and hence, fingerprints might not be necessary for such databases.

The algorithm below describes how to annotate keyed nodes of an incoming version with its key value. We assume that the document D satisfies the key constraints imposed by K which consists of q keys $(C_1, (S_1, \{P_{11}, ..., P_{1m_1}\}))$,..., $(C_q, (S_q, \{P_{q1}, ..., P_{qm_q}\}))$. We let CS_i denote the sequence of label names in C_i followed by S_i of the ith key and let $p(M)$ denote the sequence of label names on a stack M. The process of annotating keyed nodes in D can occur during the time the document is first read into memory.

Algorithm Annotate_Keys(D, K)

```
1. Initialize an empty main stack $M$.
2. Traverse the nodes of tree $D$ in document order (preorder).
   (a) Upon entering a node, push the node label onto all active stacks.
       If $p(M) = CS_i$ for some $i$, then
         initialize an empty stack $M_{ij}$ for each key path, $P_{ij}$, $j \in [1, m_i]$.
       For each stack $M_{ij}$ corresponding to key path $P_{ij}$ do
         If $p(M_{ij})$ equals the sequence of tag names in key path $P_{ij}$, then
           start memorizing $P^v_{ij}$, the XML value of current node. (**)
   (b) Before leaving a node,
       If $P^v_{ij}$ is currently being memorized and
         $p(M_{ij})$ equals the sequence of tag names in key path $P_{ij}$, then
         stop memorizing $P^v_{ij}$ (**)
         remove $P^v_{ij}$ and stack $M_{ij}$.
       If $p(M) = CS_i$ for some $i$, then
         annotate the node with key path values $P^v_{i1}$,...,$P^v_{im_i}$.
       Pop all active stacks.
```

The main stack M is used to keep track of the path from root node to the current node – the sequence of tag names from root to the current node. There are two main events during the traversal of the document in document order. Whenever a node (an open tag) is first encountered, we push the tag name of the node onto all active stacks and check if the path of the current node is equal to the concatenation of tag names in C_i/S_i of some key i. The check against C_i/S_i will tell if the node is to be keyed according to key i. If so, we create an empty stack for each key path in key i. Whenever we reach a node at the end of a key path of i, we start memorizing the value (or subtree) as we traverse the subtree of this node. Upon exit of a node (a close tag), we check if the memorization of the value has been completed. If so, we stop memorizing the value and remove the key path stack. If the current node to be exited is a keyed node, we annotate the current node with all the values of its key paths. (For the above code, a node refers to an element node and can be easily extended to handle attribute nodes. Text nodes have no labels and are not pushed onto the stacks.) Observe that all relevant key paths of a keyed node will have been visited before the keyed node is exited. Hence we are always guaranteed to have all values of the key paths upon leaving a keyed node. Moreover observe

Data	**Size**	**No. of Nodes(N)**	**Height(h)**
OMIM	27.0MB	206466	5
Swiss-Prot	436.2MB	10903568	6
XMark	11.2MB	167864	12

Fig. 7. Various statistics of our experiment data.

that there can be at most $\Sigma_{i=1}^{q} m_i$ key path stacks active at any time. The reason is that a keyed node of path l has to be exited before another keyed node of path l can be encountered. Furthermore, since we do not have keyed nodes within key path values, at most one key path value is being memorized at any time. The size of each stack is at most h where h is the height of the tree.

Analysis. We show that the running time of Annotate_Keys is dominated by the size of the document and key specification. We first analyze the statements marked `(**)`. In the extreme case, a key path value of an element may occur within the key path value of another element. Such situation occurs when we have overlapping key paths, e.g., when we have keys (/A, (B, {C/D})) and (/A/B, (C, {D/E})). The key path value under the path D/E of element C occurs within the key path value under the path C/D of element B. A naive implementation that copies P_{ij}^{v} (a key path value) can increase the size of the document by a factor of N where N is the number of nodes in the document. The total size of all key path values is $O(N^2)$ since the total size of all outermost key path values, second outermost key path values and so on is at most N, $N-1$, ..., 1 respectively. To obtain smaller key path values, we could compute the fingerprint and copy the fingerprint instead of P_{ij}^{v} (see Section 4.3). Alternatively, we have implemented P_{ij}^{v} as a pointer to the node holding the key path value. We copy the pointers around and they are subsequently dereferenced when its value is sought. For simplicity of our analysis, we shall assume that P_{ij}^{v} is implemented as a pointer to the node holding the key path value.

The algorithm scans D once, taking actions in 2(a) and 2(b) for every node. In 2(a), assuming that the time for every path comparison is proportional to the length of the path, a naive implementation for this step takes $qh + h\Sigma_{i=1}^{q} m_i$ time. The condition check for $p(M) = CS_i$ for some i takes qh time and the check that some stack M_{ij} corresponds to key path P_{ij} takes $h\Sigma_{i=1}^{q} m_i$ time. There are possibly more efficient alternatives to the implementation of the check $p(M) = CS_i$ due to overlapping paths among CS_i. See for instance [Altinel and Franklin 2000; Diao et al. 2002]. In 2(b), since we only remember the pointer to the key path value when necessary, this step takes at most $1 + h$ time where h is the time to check if $p(M_{ij})$ equals P_{ij}. It also checks if the current path is equal to CS_i for some i and performs annotations accordingly. This takes $qh + \max_i m_i$ time at most where qh is the time required to check if the current path equals CS_i for some i and $\max_i m_i$ is the time required to annotate a node with all its key path values (or pointers). Popping all active stacks takes another $\Sigma_{i=1}^{q} m_i$ time at most. Hence the total time taken for each node is $qh + h\Sigma_{i=1}^{q} m_i + 1 + h + qh + \max_i m_i + \Sigma_{i=1}^{q} m_i = O(h(\Sigma_{i=1}^{q} m_i + q))$. If D has N nodes, total time is $O(Nh(\Sigma_{i=1}^{q} m_i + q))$. Figure 7 shows the size of each parameter for the experiment data we use. Size, N and h are statistics pertaining to the largest version of each dataset.

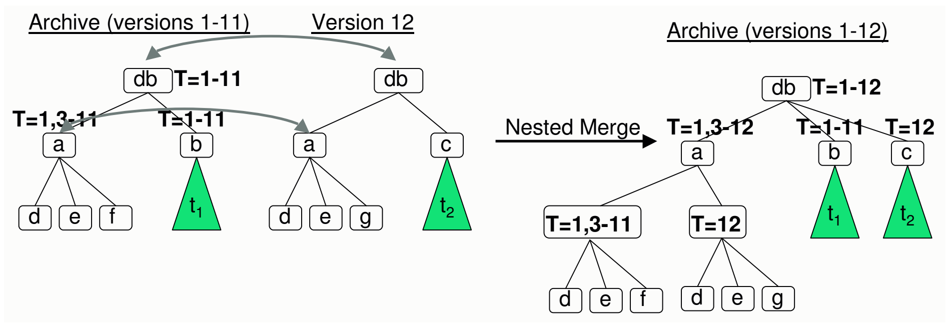


Fig. 8. Illustration of Nested Merge.

Keyed nodes of an archive are annotated in the same way. In addition, we handle the timestamp tags as follows. Whenever "...<T t=...>V</T>..." occurs where t=... is the timestamp and V represents the contents within the tag T, the root nodes of V are annotated with timestamp t=... if they are keyed. Otherwise, V must contain nodes beneath frontier nodes. In this case, V remains as a subtree of node <T t=...>.

Example 4.1. Figure 4 shows an example. Observe that the timestamp t=[3-4] is annotated on the emp{fn=John, ln=Doe} node. However, since 90K and 95K are beyond frontier nodes, they continue to exist under their respective timestamp nodes.

4.2 Nested Merge

Nested Merge is used for archiving documents where the order among keyed nodes does not matter. We first describe the main idea behind Nested Merge before describing the algorithm in detail below. The main idea behind nested merge is recursively to merge nodes in D (the incoming version) to nodes in A (the archive) that have the same key value, starting from the root. When a node y from D is merged with a node x from A, the timestamp of x is augmented with i, the new version number. The subtrees of nodes x and y are then recursively merged together. Nodes in D that do not have corresponding nodes in A are simply added to A with the new version number as its timestamp. Nodes in A that no longer exists in the current version D will have their timestamps terminated appropriately, i.e., these nodes do not contain timestamp i.

Example 4.2. Figure 8 illustrates the basic idea of Nested_Merge. An arrow between a node in the archive and a node in the version indicates that they are corresponding nodes, i.e., they have the same key value. Corresponding nodes are merged together in the new archive and their timestamps are augmented with the latest version number, i.e., number 12. Since the b element no longer exists in Version 12, its timestamp is not augmented with the latest version number in the new archive. On the other hand, since c first exists in Version 12, a new c element is created in the new archive with the timestamp t=[12]. Assuming that a is a frontier node, Nested_Merge will detect that the contents of the a element in the archive and that in the incoming version are different, i.e., <d/><e/><f/> is not the same as `<d/><e/><g/>` according to *value equality.* Hence, although there are clearly common elements among the contents, they are stored separately under different timestamp nodes in the new archive.

Value Equality. Before we describe the algorithm Nested_Merge, we shall recall the definition of *value equality* (also described in Appendix A), which is used in comparing key values of nodes in Nested_Merge. Recall that the value of a node is defined according to its type. If a node is a text node (T-node), the value is its text string. If a node is an attribute node (A-node) that has attribute name l and string v, its value is the pair (l, v). The value of an element node (E-node) consists of two things: (1) a possibly empty list of values of its E and T children nodes according to the document order, and (2) a possibly empty set of values of its A children nodes. A node n_1 is *value equal* to a node n_2, denoted as $n_1 =_v n_2$, if they agree on their value, i.e., intuitively, the trees rooted at the nodes are isomorphic by an isomorphism that is identity on string values.

In the following discussion, we assume that A and D are already annotated with keys as described in the previous section. We assume the archive A contains versions 1 through $i-1$ and D is the new version (version i), which respects key specification K, to be archived into A. To simplify discussion, we assume that elements in the incoming version or archive, except timestamp elements, do not contain attributes. The archive A is assumed to contain a single root node r_A even when no versions are present. For any node x in A, let time(x) denote the timestamps annotated on node x. Before any version is merged into A, time(r_A) exists as an empty set. Let label(x) denote the full label of a node x, including its key value but excluding timestamp annotations. For example, the label of the emp node of John Doe in Figure 4 is emp{fn=John, ln=Doe}. In general, the label of a node has the form $l(p_1 = v_1, ..., p_k = v_k)$. The labels of two nodes, $l\{p_1 = v_1, ..., p_k = v_k\}$ and $l'\{p_1 = v'_1, ..., p_k = v'_k\}$, are equal if the corresponding tag names are identical ($l = l'$) and key values are the same ($v_1 =_v v'_1$, ..., $v_k =_v v'_k$). Otherwise, they are not the same. Let r_D denote the virtual root of D where label(r_D) is identical to label(r_A) and the tree representation of D is contained as a subtree of r_D. Let children(x) denote the sequence of children nodes of x. Let xml(x) denote the XML representation of the entire tree rooted at x. We invoke the algorithm with the following arguments, Nested_Merge(r_A, r_D, {}). The last argument contains inherited timestamp, initially empty.

Algorithm Nested_Merge(x, y, T)

```
If time($x$) exists, then
  add $i$ to time($x$),
  let $T$ be time($x$).
If $y$ is a frontier node then
  If every node in children($x$) is not a timestamp node, then
    If $x \neq_v y$, then
      create timestamp node $t_1$ (i.e., <T t="$T - \{i\}$">) and attach children($x$) to $t_1$,
      create timestamp node $t_2$ (i.e., <T t="$i$">) and attach children($y$) to $t_2$,
      attach $t_1$ and $t_2$ as children nodes of $x$.
  Else
    If there exists a node $x'$ in children($x$) such that children($x'$)$=_v$ children($y$), then
```

```
            add $i$ to time($x'$).
        Else
            create timestamp node $t_1$ (i.e., <T t="$i$">) and attach children($y$) to $t_1$,
            attach $t_1$ as child node of $x$.
Else
    Let $XY = \{(x', y') \mid x' \in$ children($x$), $y' \in$ children($y$), label($x'$) = label($y'$)$\}$.
    Let $X'$ denote the rest of the nodes in children($x$) that do not occur in $XY$ and
    $Y'$ denote the rest of the nodes in children($y$) that do not occur in $XY$.
    For every pair $(x', y') \in XY$
        (a) Nested_Merge($x'$, $y'$, $T$)
    For every $x' \in X'$
        (b) If time($x'$) does not exist, then let time($x'$) be $T - \{i\}$.
    For every $y' \in Y'$
        (c) Let time($y'$) = $\{i\}$ and attach $y'$ as a child node of $x$.
```

The algorithm first determines the current timestamp. It is i added to time(x) if time(x) exists. Otherwise, the timestamp is inherited from its parent. Observe that time(r_A) always exists. It is a property of the algorithm that label(x) = label(y) whenever Nested_Merge(x, y, T) is invoked. The algorithm then proceeds by checking if y is a frontier node.

Frontier nodes are handled specially because there are no keyed nodes beyond frontier nodes. Thus the recursive merging of identical nodes no longer applies for the descendents of frontier nodes. The children nodes of any frontier node have the property that either they are all timestamp nodes or none of them is a timestamp node. In Figure 4, sal of John Doe is an example of a frontier node whose children are all timestamp nodes, tel of John Doe is a frontier node none of whose children are timestamp nodes. The transition from having no timestamp children nodes to all timestamp children nodes occurs when an incoming version is such that the content of the frontier node to be merged differs from that in the archive. Consider the last archive in Figure 9 which contains versions 1-3. When version 4 is merged, sal of John Doe in version 4 differs from that in the archive containing versions 1-3. Hence, timestamps are used to enclose the salary values at the respective times. The resulting archive is shown in Figure 4.

If y is not a frontier node, we "partition" nodes in children(x) and children(y) into three sets. (1) The first set, XY, contains pairs of nodes from children(x) and children(y) respectively with equal key values. Observe that by the property of keys, for every node in children(x), there can be at most one node in children(y) with an equal key value. (2) The second and third sets are X' and Y'. The set X' (resp. Y') consists of nodes in children(x) (resp. children(y)) where there does not exist any node in children(y) (resp. children(X)) with an equal key value.

Nested merge is recursively called on pairs of nodes in XY, inheriting the current timestamp T. To ensure that nodes of X' no longer exist at time i, timestamp T excluding i is annotated on nodes of X' if they do not already contain timestamp annotations that terminate earlier than i. Subtrees rooted at nodes of Y' are attached as a subtrees of x and they are annotated with timestamp i since they only begin to exist at time i.

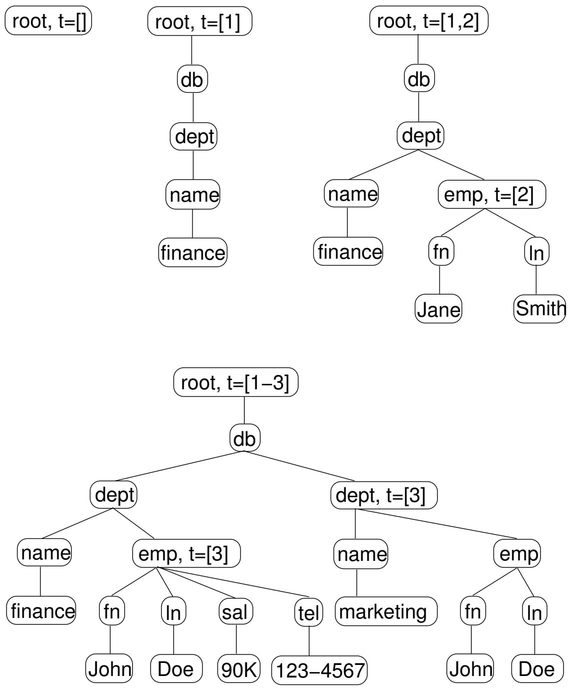


Fig. 9. States of the archive as versions 1-3 are merged. Keys of nodes are omitted.

Figure 9 together with Figure 4 shows the evolution of the archive as versions 1 to 4 of Figure 2 are added.

Analysis. We analyze the running time of Nested_Merge to show that it is $O(\alpha N \log N)$, where $N = \max(N_A, N_D)$, N_A and N_D are the number of nodes in A and D respectively, and α is the maximum time it takes to compare the labels of two nodes or test if two nodes are value equal. We assume the set of frontier paths are kept in a hash table and hashing will allow us to determine if a given path (or node) is a frontier path in expected constant time. Let d_A and d_D be the maximum degree of A and D respectively. Hence children(x) and children(y) can be determined in d_A and d_D time. According to the algorithm, if y is a frontier node, so is x. The first If statement following the condition that y is a frontier node in the algorithm executes in $O(d_A)$ time since there are at most d_A children nodes in children(x). The second If statement executes in α time. The third If statement executes in $O(\alpha * d_A)$ time assuming that the time to test whether the sequence of nodes in children(x') is value equal to the respective sequence of nodes in children(y) takes less time than testing whether $x' =_v y$, which takes α time at worst. The rest of the statements take take at worst $O(d_A + d_D)$ time to attach nodes to t_1 or t_2 respectively. Hence the statements on the condition that y is a frontier node executes in $O(\alpha(d_A + d_D))$ time.

In case y is not a frontier node, the algorithm proceeds to determine XY, X', and Y' and appropriate actions, (a), (b), and (c), are taken for nodes in each set. In fact, one does not have to determine XY, X' and Y' naively. Our implementation assumes that children(x) and children(y) are sorted in ascending order according to their key values and a merge-sort is done on the sorted nodes: Start with the first node of each sorted sequence children(x) and children(y). Call them x' and y'. (1) If label(x')$=_{lab}$label(y'), then perform action (a). (2) If label(x')$<_{lab}$label(y'), then perform the action (b) on x' and let x' be the next node in the sequence children(x). (3) Otherwise, perform action (c) on y' and let y' be the next node in the sequence children(y). We repeat this procedure until we run out of nodes on either sequence. If children(x) (resp. children(y)) is empty first, we perform action (c) (resp. (b)) on the rest of the nodes in children(y) (resp. children(x)). Since every node in A and D is sorted at some point and a merge sort is performed, Nested_Merge takes $O(\alpha N \log N)$ time.

Observe that in the description above, we have used "$\leq_{lab}$" to compare the labels of two nodes. We define "$\leq_{lab}$" for labels of nodes next. Let label(x) be $l_1(p_1 = v_1$, ..., $p_k = v_k)$ and label(y) be $l_2(p'_1 = v'_1$, ..., $p'_l = v'_l)$, where the key path values are lexicographically ordered according to p_is and p'_is. Then, label(x)$<_{lab}$label(y) if (i) $l_1 < l_2$, or (ii) $l_1 = l_2$ and $k < l$, or (iii) $l_1 = l_2$, $k = l$, $p_i = p'_i$, $v_i =_v v'_i$ for all $i \in [1, j-1]$, $j \in [1, k]$, and $p_j < p'_j$, or (iv) $l_1 = l_2$, $k = l$, $p_i = p'_i$, $v_i =_v v'_i$ for all $i \in [1, j-1]$, $j \in [1, k]$, $p_j = p'_j$, and $v_j <_v v'_j$. The label of two nodes are equal (label(x)$=_{lab}$label(y)) if $l_1 = l_2$, $k = l$, $p_i = p'_i$, $v_i =_v v'_i$ for all $i \in [1, k]$. The definition of "$\leq_v$" for XML values can be found in Appendix A. Notice that had we computed fingerprints, the process of determining whether label(x) is less than label(y) will compare the fingerprints v_j and v'_j instead of actual XML values. The use of fingerprints for ordering nodes does not affect correctness since we use the same ordering on nodes in the archive and version and all that really matters in Nested Merge is that nodes with identical key values (therefore identical fingerprints) are merged together.

Further Compaction. To obtain a further compact archive representation, one can apply diff-based techniques on values beneath frontier nodes. Within the frontier node, we represent the contents that remain the same across versions only once and mark the parts that differ by timestamps. This approach is much like the one that SCCS [Rochkind 1975] adopts. In this way, instead of representing values of these nodes under the respective timestamps, we represent only their difference. The advantage of this technique arises when values differ only slightly across versions.

EXAMPLE 4.3. Going back to the illustration for Nested_Merge in Example 4.2, Figure 10 shows the resulting archive if further compaction is applied. Observe that elements d and e occur only once in the output archive under further compaction since the difference between the values of the frontier node a in the archive and incoming version is only on the last element. As before, we use timestamps to mark the existence of the nodes at various times.

4.3 The fingerprint of an XML value

In this section, we show how one can adapt Nested Merge to use fingerprints instead of actual key values so that the version is archived correctly even when collisions of fingerprints may occur.

To compute the fingerprint of an XML value V, we first obtain the canonical

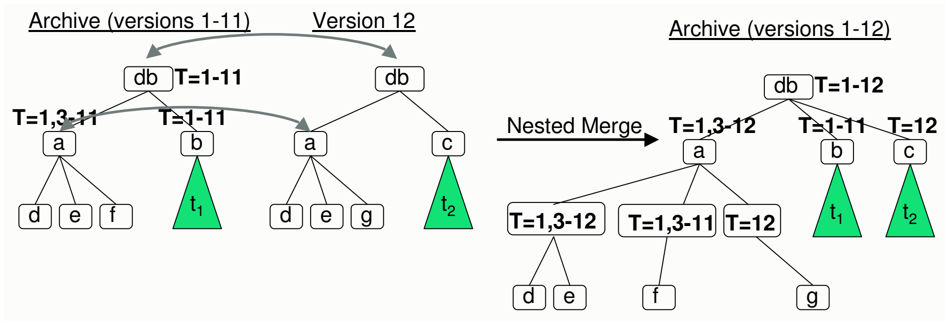


Fig. 10. Illustration of further compaction.

form of the XML value [W3C 2001a] C_V. The canonical form is such that given two XML values V and V' that are value equal, their corresponding canonical forms are string equal. That is, given two XML values V and V', $V =_v V'$ if and only if $C_V = C_{V'}$[3].

With the canonical representation of an XML value, a fingerprint of the canonical XML value can be obtained by applying DOMHash [Maruyama et al. 2000] or other hash functions (see for instance [Motwani and Raghavan 1995]) on the XML value. Therefore, given two XML values V and V' that are value equal, $f(C_V) = f(C_{V'})$ where f is the fingerprint function.

We next describe how Nested Merge can be modified to use fingerprints instead of actual key values.

Compute fingerprints. We compute the fingerprint of the key value of every keyed node in the archive and version using a fingerprint function f and canonical XML. Note that since fingerprints are applied only on key values in the archive and key values do not change, the fingerprints of each keyed node need not be updated in the archive once they are computed. Computing all fingerprints takes $O(\alpha n)$ assuming that α is the time to compute the fingerprint of the largest key value[4] and n is the number of nodes in the archive or version (whichever is bigger).

Sort. Next, we sort all keyed sibling nodes according to the fingerprints. This takes $O(\beta n \log n)$ time, assuming that it takes β units of time to compare two fingerprints. After the archive and version are sorted, we proceed to merge keyed nodes at every level according to their fingerprints.

Merge. Whenever a node in the version matches with a node in the archive (i.e., their fingerprints match), one has to verify that their actual key values indeed match. This is to ensure that we do not merge nodes with different key values. Hence, every successful match between two fingerprints in fact incurs the extra time α — the time to compare their actual key values. Assuming that the probability of error is $O(1/t)$[5], and there are at worst n nodes in the version, the expected number

[3]Although the whitespaces between elements are preserved in the canonicalization of an XML value in [W3C 2001a], our XML model ignores these whitespaces.
[4]We shall assume that the time to compare two XML key value is roughly the same as the time to compute the fingerprint of the larger key value of the two.
[5]For MD5, t is typically 2^{64} or 2^{128} which is much larger than typical values of n.

of collisions for any node in the archive is therefore $O(n/t)$. Therefore, at worst every node in the version is compared with $O(1+n/t)$ nodes in the archive during the merge process. Hence, the merge phase takes expected time $O((\beta+\alpha)(n+n^2/t))$.

We next compare the total "work" done when Nested Merge uses fingerprints with the total "work" done when Nested Merge does not use fingerprints. The total work done when Nested Merge uses fingerprints is

—**Compute fingerprints.** $O(\alpha n)$ for computing fingerprints of nodes in archive and version.

—**Sort.** $O(\beta n \log n)$ for sorting nodes in archive and version.

—**Merge.** $O((\beta+\alpha)(n+n^2/t)$ expected time for merging nodes in the version to the archive.

The total work done when Nested Merge does not use fingerprints is

—**Sort.** $O(\alpha n \log n)$ for sorting nodes in the archive and version.

—**Merge.** $O(\alpha n)$ for merging nodes in the version to archive.

Since a fingerprint is smaller than the size of the actual key value ($\beta < \alpha$) and t is typically larger than n, the total work done in either case is roughly the same. Hence the difference in total work done is essentially the time to compute fingerprints and sort nodes in the former case versus the time to sort nodes using actual key values in the latter case.

5. EXPERIMENTAL RESULTS

The main purpose of our experiments is to answer the following question: How does the storage space performance of our technique compare with traditional diff-based + sequence-of-delta approaches? Our experiments show that our archive requires only slightly more space than those existing approaches on real scientific data. Moreover, our results also show that our compressed archive is more space efficient than any compressed delta-based repository. There are many variants of delta-based approaches but we identified two likely competitive candidates: (1) the *incremental diff approach* that stores the first version and diffs of every successive pairs of versions in a repository, and (2) the *cumulative diff approach* that stores the first version and diffs of every version from the first version.

An alternative to the incremental diff approach is to store the last version together with successive backward diffs. Whether with the forward or backward diff approach, the size of the repositories should be the same since each element appears exactly once in some diff and in the first (or last) version. Thus, we focus our attention on the forward diff approach and simply refer to it as the incremental diff approach. For similar reasons, the size of a cumulative backward diff repository should be the same as that of the forward cumulative repository. Since the backward cumulative diff approach involves reconstructing all cumulative diffs whenever a new version arrives, we focus our attention instead on the forward cumulative diff approach and simply refer to it as the cumulative diff approach.

Cumulative diffs is obviously worse than incremental diffs in storage efficiency. However, it has the advantage of being fast in retrieving any version: Any version can be retrieved by one application of an edit script. Hence if the storage space required by cumulative diffs is not far from that required by incremental diffs, cumulative diffs would be a tougher competitor for us. Our experiments, however, show that cumulative diffs quickly suffers from the large storage space when the number of versions gets larger as shown later. Henceforth, we concentrate on comparisons between our archive and the incremental diff approach.

In addition to the choice of incremental diffs or cumulative diffs, we also have a choice of tree diffs or line diffs. Tree diffs have been extensively studied [Zhang and Shasha 1989; 1990; Chawathe et al. 1996; Chawathe and Garcia-Molina 1997; Cobena et al. 2001] and we used XML-Diff [XMLTREEDIFF], which is implemented for XML and is downloadable from the Web. However, when compared with line diff, XML-Diff incurred a significantly higher space overhead (and it was also not robust: We could only run it on some of our data). Since our XML data is formatted in such a way that each element is represented by one or more consecutive lines separate from other elements, line diff gives a compact representation for small changes. For this reason, we chose line diff for the comparison. In all our experiments, we used the standard unix diff command with "-d" option to compute the smallest edit scripts. Hence, the sizes of our diff repositories are always the smallest possible.

Since each version of our data is large and our basic archiver is an in-memory algorithm, we quickly ran out of memory on a machine with 256MB memory. To overcome the memory limitation, we hashed our experimental data into "chunks" based on the values of keys. An incoming version is partitioned in the same manner, and we apply our archiver to the corresponding chunks of the archive and the incoming version. Since we never merge elements with different key values, we can obtain the archive of the whole data by merging the archive and the version chunk by chunk, and concatenating the results. Since our concern was storage space, what we did to overcome the memory limitation was done in order to obtain a quick verification of how our archiving technique compares to conventional diff-based techniques. We show how to extend Nested_Merge to an external memory algorithm in Section 6 to handle large files. We note here that unix diff (line diff) also ran out of memory on large files. Therefore we applied unix diff on versions of the respective chunks as well.

We also examined how our approach and other approaches perform in combination with compression techniques. We compress the diff-based repositories with gzip (a standard file compression tool) and we compress our archives with XMill (an XML compression tool) [Liefke and Suciu 2000]. We also experimented with compressing cumulative diffs with gzip since compressed cumulative diffs may be small. As another possible competitor, we also put all the versions side by side into one XML tree and compress it with XMill. Both gzip and XMill were run with "-9" option to give the best possible compression.

In the following graphs in this section, line version shows the size of each version, and line archive shows the size of our archives storing up to each version. Line V_1+inc diffs shows the size of incremental diff repositories, i.e., the total size of the first version and all the incremental diffs, and line V_1+cumu diffs shows the size of cumulative diff repositories. Line gzip(V_1+incremental diffs) and line gzip(V_1+cumu diffs) refer to the size of incremental diff repository and cumulative diff repository

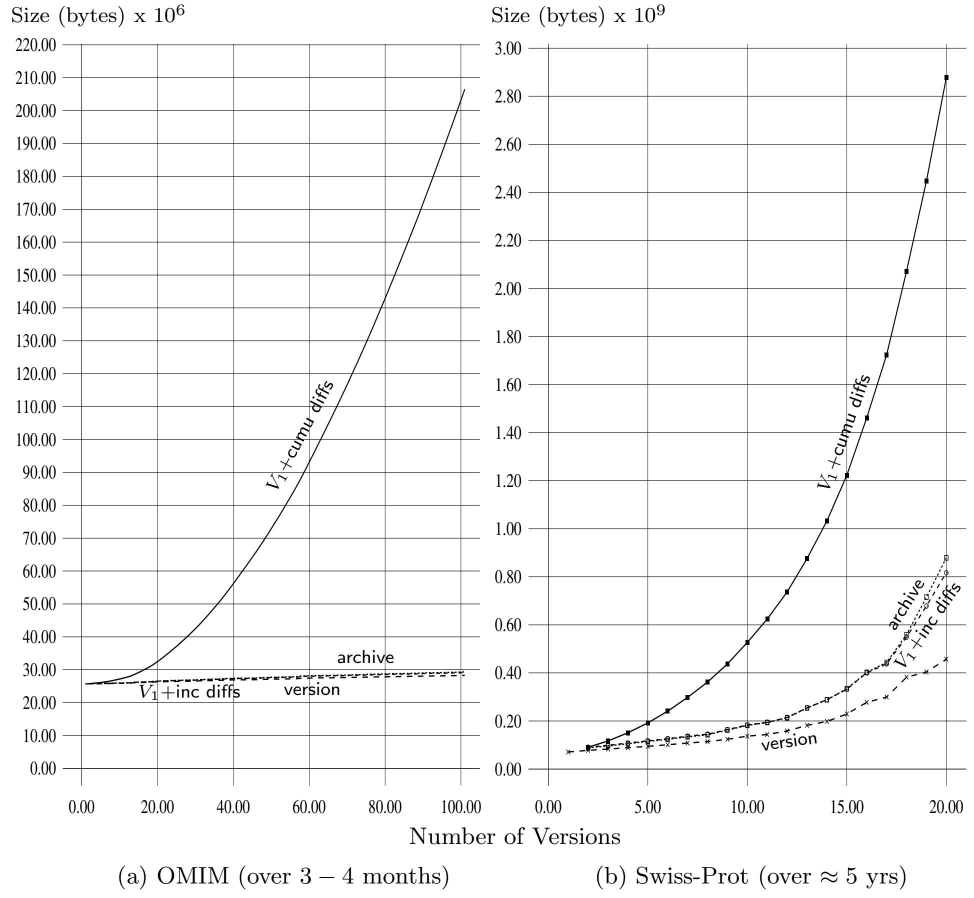


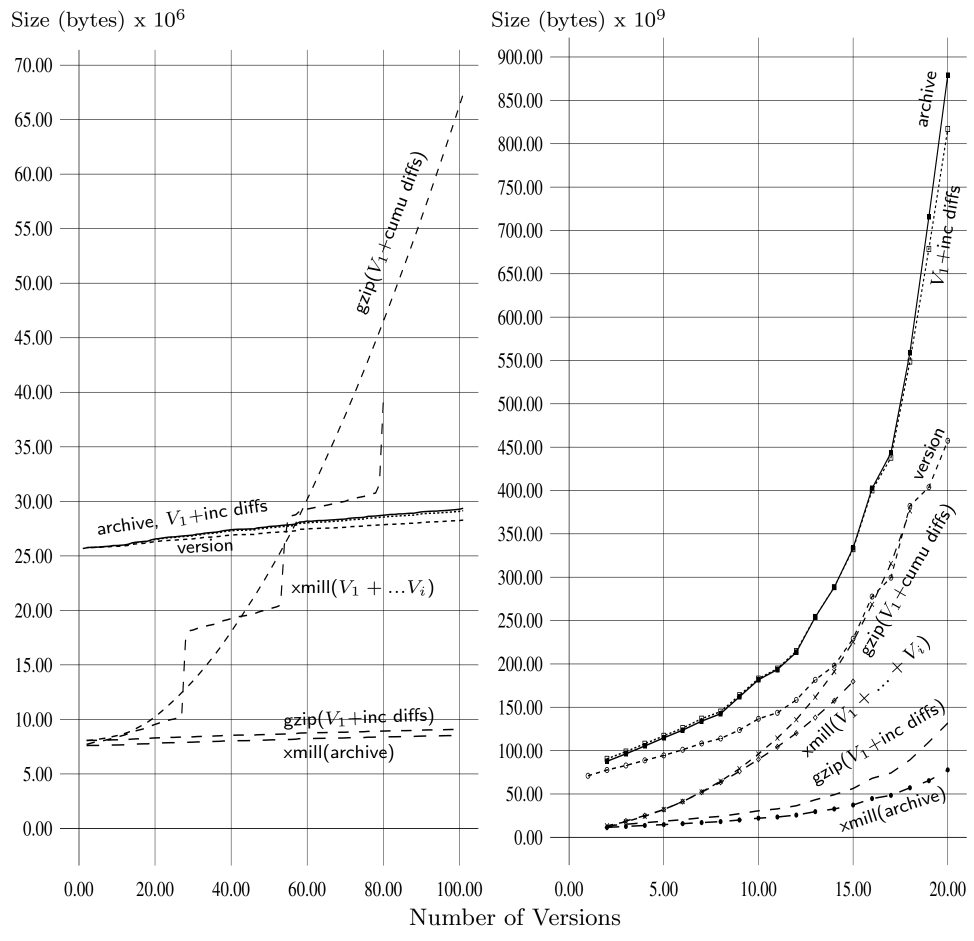
Fig. 11. OMIM and Swiss-Prot with cumulative diffs.

with gzip applied, respectively. Line xmill(archive) shows the size of our archives with XMill applied. Finally, line XMill(V_1+...+V_i) refers to the size of the concatenation of all the versions with XMill applied.

5.1 Summary of Experimental Results

We conducted experiments to compare the size of our archive with the size of diff repositories on real and synthetic datasets. For synthetic data, we simulated two types of changes; In one experiment, each version is generated with random changes with varying frequencies of change. In the other experiment, each version is generated in such a way that it gives a bad scenario for our archiver.

Our experiments show although cumulative diff repositories are attractive for their efficiency in reconstructing any past version, the space requirements grows quadratically with the number of versions. The experiments also show that the size of our archive is comparable to the size of an incremental diff repository on real datasets and synthetic data with random changes. Hence, our archiving technique is able to preserve the historical continuity of elements without incurring signifi-



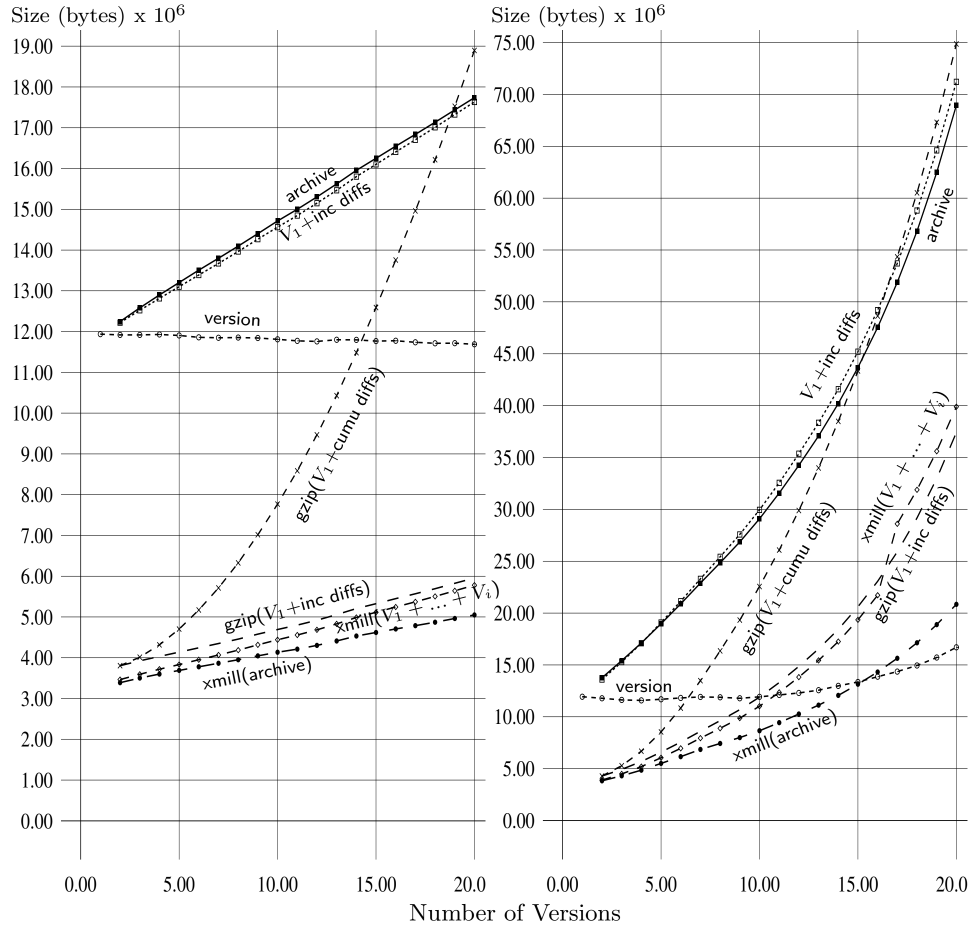
(a) OMIM (over 3 – 4 months) (b) Swiss-Prot (over ≈ 5 yrs)

Fig. 12. Storage space of OMIM and Swiss-Prot with incremental diffs.

cant space overhead when compared with diff-based repositories. Furthermore, we show that our compressed archive outperforms compressed diff repositories in all scenarios (real or synthetic). For synthetic data where we generated bad scenarios for our archiver, our archive indeed takes up much more space than an incremental diff repository as expected. Suprisingly, however, our compressed archive continues to outperform the compressed incremental diff repository even under such circumstances. Hence, if space is the main concern, we are likely to do better archiving with our technique and compressing with XML compression tools.

5.2 Comparison with Cumulative Diffs

We first show that although cumulative diff has the advantage of retrieving any past version fast, it quickly suffers from the drawback of incurring large amounts of storage space when it is applied to real data. Figure 11 shows the result of applying the cumulative diff approach to OMIM [OMIM 2000] and Swiss-Prot [Bairoch and



(a) 1.66%/1.66%/1.66% change ratio (b) 10%/10%/10% change ratio

Fig. 13. Storage performance on XMark data under different change ratio.

Apweiler 2000] data, which are real scientific data as described in Introduction. It shows that for OMIM and Swiss-Prot, the total amount of storage space required by the cumulative diff approach far exceeds that required by our archiving technique or the incremental diff based approach. When the 10th version of Swiss-Prot is added, the cumulative diff repository already incurs more than twice the storage space used by our technique or the incremental diff approach. Therefore, we concentrated our experiments on comparisons between storage costs of our archive and incremental diff repositories.

5.3 Comparison with Incremental Diffs

The graphs in Figure 12 again show the results of the experiments with OMIM and Swiss-Prot but without the line for cumulative diffs (and with some additional lines explained later). In those graphs, we see that the incremental diff approach marginally outperforms our approach for most cases. For OMIM, the archive size

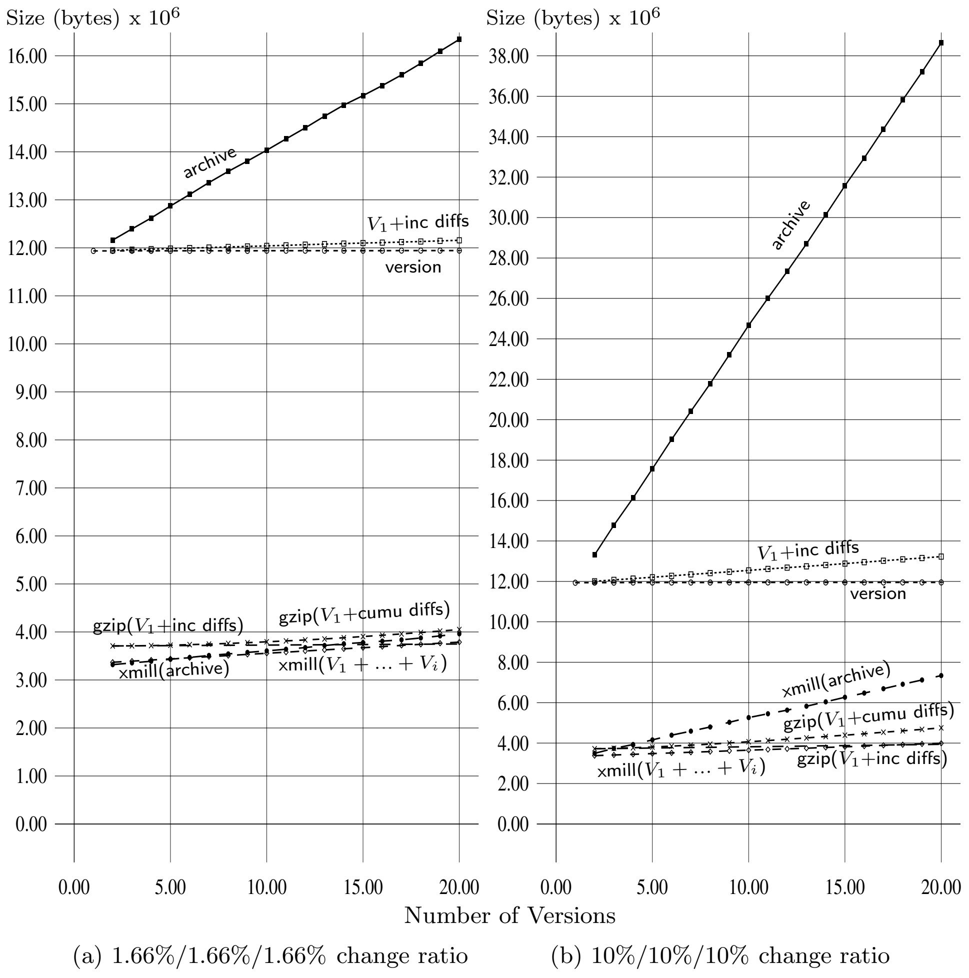


Fig. 14. Storage performance on XMark data with insertion and deletion of highly similar elements.

is never 1% more than the size of incremental diffs, and for Swiss-Prot, the archive size is never 8% more than the size of incremental diffs.

In the graph for Swiss-Prot, the size of our archive grows quadratically or even faster. This is because the size of each version grows very fast, and as it grows, the amount of change between two consecutive versions also grows. Compared with the incremental diff approach, which logically achieves the smallest space cost, our approach is not doing much worse.

Observe that the data stored in our approach and the diff approach are essentially the same (although they are organized in different ways, i.e., grouped by elements versus grouped by time). The small difference between the storage space in the two approaches mainly comes from the difference between the size of timestamps in our archives and the size of "markers" in diff scripts. If we have a huge number of changes of very small data, such as a text data of length 1, the ratio of that difference to the size of the archive can be considerable. Such an extreme situation, however, rarely occurs in the experimental data we used.

Although Swiss-Prot and OMIM were recorded for different periods and at different intervals, it is worth noting that in both cases the space required for the archive diverges from the space required for the most recent version at about 15% per year. This is a crude measure of the rate of change in the database, and it would be interesting to know if other curated databases have a similar rate.

To make a comparison between our approach and the diff-based approach under various change ratio to the data, we also produced synthetic data sets from XMark data [Schmidt et al. 2002] under various change ratio. The first version of our database is the synthetic auction data from XMark. Our change simulator creates a new version by deleting n% of elements, inserting the same number of elements with random string values, and modifying string values of n% of elements to random strings. By repeatedly applying our change simulator, we generated 20 versions with various values for n.

The graphs in Figure 13 shows the results of the experiments for $n = 1.66$ and $n = 10$. For reference, the deletion/insertion/modification ratios between two versions of OMIM and Swiss-Prot are roughly 0.02%/0.2%/0.03% and 14%/26%/1.2%, respectively. Various other statistics of these data can be found in Figure 7. In the graph for $n = 1.66$, i.e. 1.66% insertion + 1.66% deletion + 1.66% modification, the diff approach marginally outperforms our approach again. On the other hand, in the graph for $n = 10$, our approach marginally outperforms the diff approach. The main reason for the improvement is that our change simulator modify string values to random strings, and when the ratio of the modification is high, a text sometimes happens to be modified to some of its old values. While incremental diff approach is forced to store those values multiple times in different deltas, our approach stores the value only once, and timestamps are augmented appropriately. Clearly, the former incurs a higher overhead than the latter in most cases.

On the other hand, when highly similar data are deleted and inserted at the same location, the diff-based approach has the advantage. The diff-based approach stores only diffs between those two similar elements, while our archiver stores those two elements separately because their key values are not the same. The graphs in Figure 14 shows the results of experiments simulating that worst-case scenario for us. Here, we used XMark data, but our change simulator modifies part of key values for n% of elements instead of deleting and inserting n% of elements. It is simulating deletion and insertion of highly similar data at the exactly same location. As shown in the graphs, the size of our archives grow very fast in this worst scenario, while the size of the diff repository is only slightly bigger than the size of one version.

By extrapolating the scenario above, one can see that in the extreme case where the same data is repeatedly deleted and inserted, our approach is likely to outperform incremental diff approach by a big margin. On the other extreme, there may be insertions and deletions of highly similar data (though semantically different because they have different key values) over the versions. In this case diffs usually have a compact representation (by storing only the difference between them) while our approach is forced to store the highly similar elements separately. Both such extreme cases, however, rarely happens in scientific data. With typical scientific data which are highly accretive, i.e., data where most of changes are addition, the size of our archive and the size of the diff-based repository would be roughly the same. The experimental results with OMIM data, changes to which is mainly addition as shown in Figure 7, verifies that observation.

5.4 Interaction with Compression

We have seen that incremental diff technique may outperform our technique in terms of storage space. However, when compression is applied to both our archives and incremental diff repositories, our archives use less storage space in most of our experiments. For both OMIM and Swiss-Prot (in Figure 12), we see that XMill applied on our archive (xmill(archive) line) outperforms any other approaches, i.e. gzip(incremental-diff), gzip(cumulative-diff), and xmill($V_1 + \ldots + V_i$), even during the times when the incremental diff repository is clearly smaller than our archive without compression. For instance, in Figure 12(b), as our archive starts to perform worse than the incremental diff repository after Version 17, our compressed archive continues to outperform the compressed diff repository by a growing margin. By the time Version 20 of Swiss-Prot is added, the storage space used by our archive is clearly more than incremental-diff. However, gzip(incremental-diff) uses clearly more storage space than xmill(archive) by about the same magnitude. It seems the margin between gzip(incremental-diff) and xmill(archive) is bigger when the change ratio is high. It is bigger in Figure 12(b) than in Figure 12(a), and within Figure 12(b), it is bigger at Version 10...20 than at Version 1...10. The same characteristic can also be seen in Figure 13.

This reversal of storage space efficiency occurs because our archive of XML data sets is again in XML format, and XMill can exploit this specific structure to obtain a better compression ratio than general compression tools like gzip. XMill groups text data according to the names of the elements in which they occur and compresses each group separately. Since text data that belong to elements of the same name tend to be fairly similar, high compression ratios can usually be achieved.

Higher compression ratio of xmill(archive) can be seen even in the graphs of the worst-case scenario in Figure 14. In those graphs, xmill(archive) is smaller than gzip(incremental-diff) up to the points where our archive gets about 1.2 times larger than the incremental diff repository (ver. 14 in (a) and ver. 3 in (b)). After those points, the difference of the size of our archive and the diff repository seems outrun the higher compression ratio of XMill.

Our graphs also show that a simple application of XMill to the concatenation of all the versions does not work as well as our archive compressed with XMill (see lines xmill($V_1 + ... + V_i$) and xmill(archive)). Finally, the combination of gzip and cumulative diffs achieves a rather high compression ratio. However, it still does not perform as well as the XMill applied to our archive.

Remarks. In Figure 12(b), we reached the system file size limit of 2^{31} when trying to add Version 19 to V_1+cumulative-diffs repository before applying gzip. The limit is also reached when trying to add Version 16 to the $V_1 + ... + V_{15}$ repository before XMill can be applied and hence the discontinuation in lines gzip(V_1+cumulative-diffs) and xmill($V_1 + ... + V_i$) in this graph. In the same way, in Figure 12(a), xmill($V_1 + ... + V_{80}$) reaches the limit. In addition, the stepping in the xmill($V_1 + \ldots + V_i$) is due to the fact that xmill is using "dictionary compression" and is increasing the space allocation for the dictionary.

6. EXTERNAL MEMORY NESTED MERGE

The algorithm Nested_Merge, presented in Section 4.2, is an "in-memory" algorithm. Nested_Merge assumes that the entire archive and version can be read into memory before deciding which nodes can be merged. The smallest version in Swiss-Prot has about $2,037,292$ nodes and can contain as many as $10,903,568$ nodes. On a machine with a standard amount of memory, Nested_Merge clearly cannot handle such files. To overcome the memory limitation, one solution is to use PDOM (Persistent Document Object Model) [PDOM]. PDOM provides DOM API to process XML documents that far exceed memory limits. However, the performance of PDOM largely depends on how the XML file is fragmented, what indices are built according to PDOM implementation and how the XML file is accessed.

In this section, we describe how one can extend Nested Merge to an external memory algorithm to handle data where the size of data may be much larger than the available memory. The main steps taken by the external memory archiver, which we outline here, are essentially the same as that described Section 4.

External Memory Archiver

Input: An archive, a new version (to be merged into the archive), and a set of key specifications.
Output: A new archive that contains the new version.

(1) **Annotate nodes with their key values.** Both the archive and new version are preprocessed so that, conceptually, every keyed node has its key value immediately associated with it. (See Section 6.1.)
(2) **Sort archive and new version.** The archive and new version are each sorted according to keys, i.e., all sibling nodes are ordered according to key values. (See Section 6.2.)
(3) **Merge sorted archive with version.** The archive and version are merged together by making a single pass through both documents. (See Section 6.3.)

In the ensuing description, we let M be the total memory size and B be the size of a page (or 1 disk block). Let the total size of a version or archive (whichever is larger) be N. We are of course interested in the case when $M < N$. We assume that every root-to-leaf path (including all key values of nodes along the path) in our document can fit in one page. This means $h < B$ (which is generally true in practice), where h is the height of the document. For example, the heights of OMIM, Swiss-Prot and XMark versions are 5, 6 and 12 respectively. The heights of the respective archives is at worst twice (due to timestamp tags) the heights of the respective versions.

6.1 Annotate nodes with their key values

Prior to sorting a document according to key values, we first annotate every keyed node with its key value. Our implementation makes use of an internal representation of an XML document similar to that used by XMill [Liefke and Suciu 2000]. This step breaks an input document into three main types of files for subsequent processing: (1) an internal representation of the document, (2) a dictionary, and (3) keys files.

The internal representation of the document is a document with tag names replaced by integers. In addition, several bits are used to indicate whether this element is an open or close tag, and whether it has a key value, attributes, or timestamp if it is an element node. We create an internal representation of the XML document as we scan through the document. Whenever a new tag name is encountered, it is assigned a new number. We represent a tag number with two bytes in our internal representation and thus storing tag numbers may use less space than storing tag names. In addition, some bits are used to store the additional information. The key value of a node is fully determined by the time that node is exited. If the root-to-node path is p then the key value is appended in file p. Hence, there are as many key files as the number of keys in the set of key specifications. The dictionary file consists of the mapping between tag names and numbers and the same file is used for both the archive and version.

Example 6.1. For example, version 4 of Figure 2 is decomposed into the following files (we omit the marketing department):

Internal representation (";" represents a close tag):
1 2 3 "finance" ; 4 5 "John" ; 6 "Doe" ; 7 "95K" ; 8 "123-4567" ; ; 4 5 "Jane" ; 6 "Smith" ; 7 "95K" ; 8 "123-6789" ; 8 "112-3456" ; ; ; ;

Dictionary:	Key files (keys with empty key paths are omitted):
1 ↔ db	/db/dept:
2 ↔ dept	name=finance
3 ↔ name	/db/dept/emp:
4 ↔ emp	fn=John ln=Doe
5 ↔ fn	fn=Jane ln=Smith
6 ↔ ln	/db/emp/tel:
7 ↔ sal	123-4567
8 ↔ tel	123-6789
	112-3456

Assuming that key path values do not overlap, which is true in all the practical data we used in our experiments, the size of all key files is proportional to the size of the original document. The total size of the internal representation, key files, and dictionary is thus proportional to the size of the original document, it is easy to see that this step incurs an I/O cost of $O(N/B)$ for reading and writing the document.

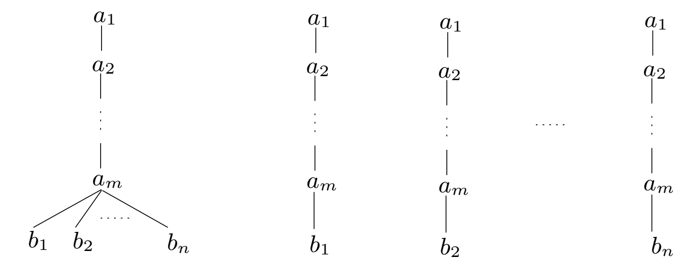
6.2 Sort archive and new version

To sort a document, we first create sorted runs of parts of the document [6]. We read the internal representation of the document in document order until we reach the memory limit M. Whenever a keyed node is encountered, we read in the appropriate key values from the relevant key file. When we reach the memory limit, sort the partial tree in memory and write it out to disk (i.e., a sorted run). At every keyed level, the nodes at that level are sorted according to their key values. Nodes beyond the frontier level are not sorted. Repeat the above process until the entire document has been parsed. Observe that at this stage, we use only the internal representation

[6]We note here that there is also ongoing work on sorting large XML files [Avilla-Campillo et al. 2002]. However, it is not clear that the work is immediately applicable for our needs.

and key files of the archive or version. Moreover, we only make a single pass through these files to create the sorted runs.

In the course of producing sorted runs, many root-to-leaf paths may be duplicated across the runs. Hence, the total size of all sorted runs may be larger than the size of the original document. For example, consider a document which has a long stem $a_1, ..., a_m$ and many leaves $b_1, ..., b_n$ as shown below.



Original document Run 1 Run 2 Run n

It may happen that the memory limit is reached whenever a b_i is read (Recall that we assume every root-to-leaf path can fit in a page.). Therefore, each sorted run produced consists of $a_1/.../a_m/b_i$. The stem $a_1/.../a_m$ is copied n times across each sorted run. Since, in the worst scenario, each sorted runs begins with h units of memory already occupied by some root-to-leaf path (the point where the algorithm stops in the previous run), the number of sorted runs is bounded by $O(N/(M-h))$. Due to duplication, the total size of all sorted runs may have increased by $O(hN/(M-h))$. Hence we have used a total of $O(N/B + hN/(B(M-h)))$ I/Os so far.

To obtain a sorted tree, we repeatedly merge the sorted runs until we end up with one sorted run containing the entire tree. We start by placing a marker at the root of each $(M/B)-1$ trees in memory. Consider the markers at height 0 (hence all markers initially), we output the marker node with the smallest label. Next, for every marker node whose label equals the smallest value, we advance the marker to the next node of the tree (in document order). Observe that some markers will advance to their siblings while other markers will advance to their children nodes. We repeat the process by considering markers occurring at the deepest level in all runs. Everytime the output buffer is full, we simply write it out to disk and keep the path from the root to the last node we emitted. During the merge process, whenever a frontier node is encountered, we write out the subtree in the order which the runs are created. In this way, we ensure that the relative order of unkeyed nodes remains unchanged.

Since there are $O(N/B+hN/(B(M-h)))$ number of pages in all sorted runs, each merge phase of $(M/B)-1$ sorted runs can be done with $O(N/B+hN/(B(M-h)))$ I/Os. With $N/(M-h)$ sorted runs at most, we have roughly

$$\begin{aligned}\log_{(M/B)-1} N/(M-h) &= \log_{(M/B)-1} N/B - \log_{(M/B)-1}(M-h)/B \\ &\leq \log_{(M/B)-1} N/B\end{aligned}$$

phases ($h \leq B$ and $M > 2B$). Therefore we use a total of $O((N/B + hN/(B(M-h))) \log_{(M/B)-1} N/B)$ I/Os.

6.3 Merge sorted archive and version

This step is very much like the previous step except that frontier nodes are handled differently — in a way similar to that described in the basic Nested_Merge algorithm. Given an archive A and version D, the external memory nested merge algorithm first sorts A and D using the sorting procedure above. Call the sorted files A' and D' respectively. We let x and y denote nodes in A' and D', respectively. Initially, x is the root of A' and y is a virtual root of D' with the same key as x and x and y proceed through A' and D' in document order. We keep track of the timestamp of the current node in A' as we traverse the document. If label$(x) <$ label(y), we output x and its entire subtree and attach the current timestamp to x. If label$(x) >$ label(y) we output y and its entire subtree and attach timestamp i (the new version number) to y. Otherwise label$(x) =$ label(y). In this case, we output x and attach the current timestamp together with i if a timestamp originally exists at x. Otherwise we output only x. If y is a frontier node, we proceed in a similar way as described in Nested_Merge algorithm in Section 4. Subsequently, we move x and y to the next nodes of A' and D' in document order and repeat the procedure. If we reach the end of A' first, we output the entire subtree rooted at y with timestamp i and repeat with the next y until we reach the end of D'. If we reach the end of D' first, we output x and its entire subtree together with the existing timestamp if any. We let x denote the next node in A' and repeat until we reach the end of A'. Since this step makes one pass through the archive and version, it incurs another $O(N/B)$ I/Os.

We remark that a line-diff algorithm could also be extended so that it can handle large files. Given two versions that each far exceeds the available memory, a line-diff algorithm can operate on "windows" of text, i.e., break each version into pages and compute diff on corresponding pages of data of the two versions. Although the diff result may not be optimal in this way, this method uses only $O(N/B)$ I/Os.

7. EFFICIENTLY RETRIEVING A VERSION OR TEMPORAL HISTORY

We show how to retrieve a version and the temporal history of a keyed element efficiently. Our archive can already support these basic operations efficiently without building any index structures. Here, we show that the complexity of these operations can be further reduced by defining simple index structures on our archive: A version can be retrieved in time roughly proportional to the size of the version and the temporal history of a keyed element can be retrieved in time roughly proportional to the size of its key.

7.1 Retrieving a Version

The structure of the archive is such that a simple scan through the archive can retrieve any version. Whenever a timestamp tag is encountered, we output the contents of the timestamp element if the required version number can be found in the timestamp. This process is likely to be more efficient than the sequence-of-delta approach whereby reconstructing a version may require many scans over the data in order to apply all the relevant deltas. Our approach, however, still takes time proportional to the size of the archive even when the size of the version to be retrieved is much smaller than the archive. We describe here a simple scheme that

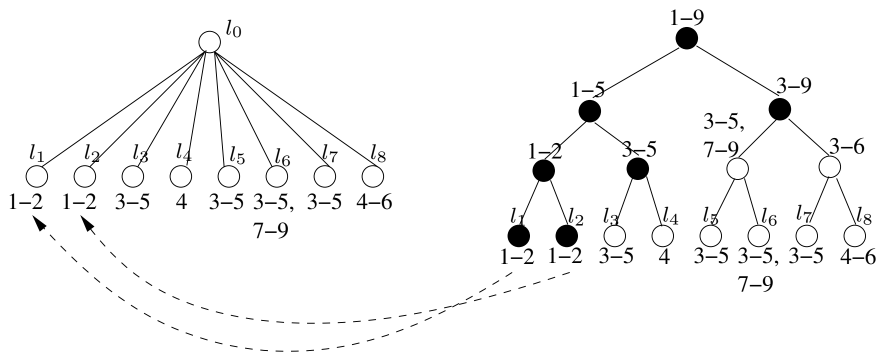


Fig. 15. An archive (shown on the left) with timestamp tree of node l_0 (shown on the right).

allows such retrievals to be performed in time roughly proportional to the size of the version to be retrieved using auxiliary timestamp binary trees.

Timestamp Trees. The purpose of a timestamp tree is to direct the search for the relevant nodes of a version in the archive. We will assume a timestamp tree for each non-timestamp node (node that is not representing the timestamp element) in the archive. Given a node x in the archive with k children, the corresponding timestamp tree is a binary tree with k leaves, one for each child of x. Every leaf of the tree has two kinds of information: a timestamp and an offset to the corresponding child node in the archive. Each internal node of the tree consists of the union of timestamps of its children nodes. Figure 15 shows a timestamp tree corresponding to the root node l_0 of the archive shown on the left. For leaf nodes, the offset of the corresponding nodes in the archive are not shown. We will describe how to construct this binary tree later.

Retrieval. We wish to retrieve a version i from the archive in time roughly proportional to the size of the version. Consider a node x in the archive and suppose it has k children of which only α belong to version i. The naive approach checks each of the k nodes to see if it is relevant to version i, i.e., has timestamp i. With timestamp trees, we first look for the timestamp tree corresponding to node x through the offset in x. Then we search down the the timestamp binary tree by starting at the root r of the binary tree and check if $i \in \text{time}(r)$ (where time(x) denotes the timestamp of node x). If $i \notin \text{time}(r)$, we stop. This means that neither of the two children nodes of r is relevant for version i. If yes, we recurse on the two child nodes to continue the search. We repeat this search down the tree. Assuming that we have a complete binary tree and k and α are powers of 2, at most α nodes will be searched in each level after depth $\log \alpha$. Once we reach the leaf nodes, we will know the relevant α nodes in the archive through the offset values stored at the leaves in the binary tree. As we are making the search down the tree, we also keep track of the number of nodes we have searched in the binary tree so far and stop at the point when we have searched a total of k nodes. If the threshold of k is reached before reaching the leaf nodes, we simply search all k leaf nodes. Observe that if $i \in \text{time}(r)$ then $\alpha \geq 1$.

An example is shown in Figure 15. Suppose we wish to retrieve version 2 from the archive shown on the left of the figure and suppose one already knows l_0 contains timestamp 2. It remains to figure out which of the children nodes l_1, ..., l_8 in the archive are relevant for time 2. We first check if the number 2 is in the timestamp associated with the root node of the timestamp tree (shown on the right of Figure 15) corresponding to l_0, i.e., $1-9$. Since $1-9$ contains the number 2, the search continues down its children nodes. The left child node contains timestamp 2 and the search continues down its subtree. However, the right child node (with timestamp $3-9$), does not contain time 2. Hence, the search stops at that node. The nodes that are searched are highlighted in the timestamp tree. Using the offsets at the leaf nodes, one can pull out the relevant nodes, i.e., l_1 and l_2, from the archive.

With this strategy, we will either probe at most $2\alpha - 1 + 2\alpha \log(k/\alpha)$ nodes (the maximum number of nodes probed when we search all the way down to the leaves of the timestamp tree) or at most $2k$ nodes of the binary tree (when we search k nodes before reaching the leaves of the tree). We search at most $2\alpha - 1 + 2\alpha \log(k/\alpha)$ nodes when $\alpha \leq k/8$ and at most $2k$ nodes when $\alpha > k/8$. In the former case, we are at worst a factor of $\log k$ away from the optimal number of probes while in the latter case, we are a constant factor away from the optimal. In largely accretive data where data changes infrequently, it is usually the case that $\alpha > k/8$ when a version is retrieved.

Constructing Timestamp Trees. Given an archive, one can construct a timestamp tree for each non-timestamp node with a single scan through the archive. The idea is to first collect the k children nodes of a node x along with their respective timestamps and offsets into the archive file. We build these k leaf nodes into a binary tree by pairing nodes repeatedly in a bottom-up manner and taking the union of timestamps of the children nodes for each internal node. Finally we append the binary tree to another file and write the offset to this binary tree in node x of the archive. Internal nodes of the binary tree will contain offsets to their children nodes in order for one to skip to the relevant nodes of the binary tree during the search. The timestamp trees are created each time a new version arrives and after nested merge is applied.

The benefit of using timestamp trees over a simple scan of the archive to retrieve a version depends on many factors. If retrievals are rare and the version to be retrieved is about the same size as the archive, then it may not make sense to use timestamp trees. On the other hand, if retrievals happen often and the version to be retrieved occupies only a small portion of the archive, then it may pay off to use timestamp trees to look for the relevant nodes. The tradeoff depends not only on the workload and data characteristics but also largely on how timestamp trees are laid out on disk. A detailed experimental analysis is beyond the scope of this paper.

7.2 Retrieving the Temporal History of a Keyed Element

Given the key of an element, one might like to retrieve the temporal history of this element, i.e., the times at which this element exists. For example, the history of employee Joe given by the path /db/dept[name=finance]/emp[fn=John, ln=Doe], is 3,4.

We sketch a naive scheme that allows one to answer this query in time $O(l \log d)$, where l is the length of the key and d is the maximum degree of the archive. The idea is to maintain, for each keyed node in the archive, a sorted list of key values of children nodes. Each node in the sorted list contains its key value, an offset to its sorted list of children nodes (index offset), and an offset to the timestamp node in the archive in which its timestamps are inherited from (timestamp offset). We assume that these information are kept in fix size records.

Using the example archive from Figure 4, there is a sorted list of children nodes, dept[name=finance] and dept[name=marketing] for the node db. There is also a sorted list emp[fn=Jane,ln=Smith], emp[fn=John,ln=Doe] for node dept[name=finance] and so on. To retrieve the temporal history of the element /db/dept[name=finance]/emp[fn=John, ln=Doe], we first use the sorted list of the db node. A binary search for the node dept[name=finance] is performed on the list. The index offset in the record dept[name=finance] will lead us to the second sorted list. A binary search for John Doe is then performed on the second list and the timestamp offset in the record of will lead us to the desired temporal history.

It is easy to see that to retrieve the temporal history of an element given by a path, one performs a binary search for every key along the path. If a key along the path is not found in the sorted list, the required element does not exist in the archive. Otherwise, we locate the next index list and repeat our binary search. At the end, we use the timestamp offset in the record of the desired element to locate the timestamp node in the archive. The process therefore takes $O(l \log d)$ time. If a node has a large number of children nodes, one can also consider building more sophisticated index structures, such as a B+ tree, for these children nodes.

The sorted lists can be constructed with a single scan through the archive. The idea is to first collect the key values of all children nodes of each keyed node as we scan through the archive, keeping track of the timestamp offsets as we go along. All key values of children nodes of any node x are known by the time x is exited. The key values are then sorted and the offset to this sorted list can be written as the index offset in the record corresponding to node x.

Like timestamp trees, the benefit of using an index over a simple scan of the archive to retrieve the temporal history of a keyed element depends on many factors. If retrievals are rare and the archive is small, then it may not make sense to create or use an index. On the other hand, if retrievals happen often and the archive is huge, then it may pay off to use the index to pinpoint the desired node quickly. A detailed experimental analysis is beyond the scope of this paper.

8. RELATED WORK

There has been considerable research on version management systems for hierarchical data recently [Marian et al. 2001; Chien et al. 2001; Chawathe et al. 1998]. Xyleme system [Marian et al. 2001] takes the sequence-of-delta approach. They store the latest version together with all forward completed deltas – changes between successive versions that can also allow one to get to an earlier version by inverting deltas on the latest version. Additional information has to be kept in order for deltas to be invertible. Another approach taken by Chien et al. [Chien et al. 2001] is a reference-based versioning scheme. The first version is always fully stored. As subsequent versions arrive in the form of updates describing the changes, only newly inserted objects in the latest version are stored. Objects that remain unchanged are stored as references to the corresponding object in the previous version. It is shown that this approach provides good support for (temporal) queries since it preserves the logical structure of every version as opposed to diff-scripts. Moreover, a usefulness-based storage scheme is shown to give good storage cost and supports efficient reconstruction of any past version. An earlier versioning scheme proposed by Chawathe et al. [Chawathe et al. 1998] stores updates as annotations on nodes and edges of a graph in the DOEM model as edits are made. A specialized query language Chorel where annotations can be queried is used to query such changes.

There has been substantial work on diff-based approaches for tree-like structures [Tai 1979; Zhang and Shasha 1989; 1990; Chawathe et al. 1996; Chawathe and Garcia-Molina 1997; Cobena et al. 2001]. The general problem of finding the minimum cost edit distance between two ordered trees was first studied by [Tai 1979] and subsequently improved by Zhang et al. [Zhang and Shasha 1989; 1990]. Chawathe et al. [Chawathe et al. 1996] made further restrictions to obtain a faster algorithm for the minimum cost edit distance. A heuristic change detection algorithm for unordered trees is also proposed by [Chawathe and Garcia-Molina 1997] with a richer set of edit operations. Another diff technique is proposed by [Cobena et al. 2001] that uses signatures to find matchings and another top down phase to compute further matchings heuristically. An external memory algorithm for computing minimum-cost edit script for two ordered trees has has been proposed by [Chawathe 1999]. Recently, the use of keys in computing diffs has also been proposed by DeltaXML [Fontaine 2001] as a more systematic way for identifying corresponding elements between two XML versions. In DeltaXML, if some element has a special attribute key (in their namespace), the system uses the value of key attribute for identifying corresponding elements in different versions. The key attribute and its value has to be created manually or automatically, e.g. by using XSL. Moreover since key values must "fit" into the unary attribute key, it suffers from the same problems as ID attributes of XML standard in expressing composite keys. Software configuration management systems such as CVS [CVS] use diff-based techniques that store the last version together with backward deltas based on line-diffs [Miller and Myers 1985; Myers 1986]. Our archiving system is more like the Source Code Control System (SCCS) [Rochkind 1975], where the (non-)existence of lines of text at various versions are indicated with timestamps. In SCCS, a line-diff is performed between the incoming version and the latest version in the repository to determine the forward delta (deletions and insertions only) from the latest version to the new version. The repository file is then updated according to the delta. A single scan through this repository file retrieves any version. However, changes are not grouped by identical lines. If the same line is repeatedly inserted and deleted over time, the repository file will contain a series of identical lines marked as deleted and inserted. With keys, our archive will only contain the line once and we use timestamps to mark the existence of the lines at the various times. Nested Merge bears a resemblance to the deep union operator introduced by [Buneman et al. 1999] in that it is conceptually taking a union of two deterministic trees. It is also similar to the Merge Template of [Tufte and Maier 2001] that merges two XML documents according to a specified template. The template specifies which elements can be merged together, inserted or replaced.

Our work is largely inspired by the work of Driscoll et al. [Driscoll et al. 1989] who studied methods for making data structures persistent. A data structure is *partially persistent* if it supports updates to the latest version and access to past versions. Linked structures where each node is assumed to hold a set of field name and value pairs can be made partially persistent by the *fat node* method. Each node holds a timestamp of when it is first created and field values within "inherit" this timestamp. Whenever a field value within a node is modified, the new value along with its field name and the current timestamp are added to the node. An access to version i of field f is done by finding among the values of f, the value with the largest timestamp less than i. Our data structure is that of a key-annotated tree where each node has a label and outgoing nodes contain distinct labels due to keys. One can view the set of child labels of a node as a set of field names in the fat node. However in our case, each field value can be yet another set of field names, and so on. Another difference concerns updates. Driscoll et al. [Driscoll et al. 1989] assumes the updates to the latest version are known and hence one is not required to compute the diff. In our case as well as others [Chawathe et al. 1996; Chawathe and Garcia-Molina 1997; Marian et al. 2001; Cobena et al. 2001], the input is yet another version and hence one is required to compute the correspondences (or difference). One can certainly apply updates to the latest version directly on the archive if known.

Our technique differs from existing versioning schemes in the several ways. First, elements in every version are identified by key. A key for an element gives a meaningful identifier to the element and is naturally persistent as opposed to generated object identifiers. With such persistent identifiers, one can easily detect identical elements across versions, thus arriving at meaningful change descriptions. Diff techniques are based on minimizing edit costs which may give semantically meaningless deltas as seen by the Person(Name, BirthDate, Address, Zip) example in Figure 1 of Section 1. Second, the sequence-of-delta techniques store many deltas separately. Apart from the ability to quickly answer queries about the difference between two successive versions, sequence-of-delta approaches do not immediately lend themselves to support other (temporal) queries such as the evolutionary history of an element during a specified time period. Third, although the work by [Chien et al. 2001] supports temporal queries to some extent through special procedures tied to their storage scheme and Chawathe et al. [Chawathe et al. 1998] supports temporal queries through a specialized query language, they assume that edits that generate each new version are available. In some cases, edits are not always available. Although OMIM and Swiss-Prot publish update logs tracking new entries and entries that have been modified since the last version, they do not publish exact changes made to each modified entry. It is also not clear whether they keep track of these changes. Our technique does not assume the existence of edit scripts and still makes it easy to discover the evolutionary history of any element. Moreover, since our archive is in XML, existing XML query languages such as XQuery [W3C 2001b] can be used to query such documents. It is also mentioned by [Chien et al. 2001] that the ideal solution is to represent the history of a versioned document as yet another XML document, for such representation will leverage on web-browsers, style sheets, query processors and other tools that already support XML.

Most work on temporal databases are in the relational context [Torp et al. 2000]. Every tuple has a timestamp that indicates the period of time that tuple exists. If a non-key attribute of some existing tuple is modified, a new tuple with the current timestamp is created. The new tuple has the same values on all attributes of the tuple whose values are unmodified and contains the new value only on the attribute whose value has been changed. In contrast, our approach pushes timestamps as far down the hierarchy as possible. Therefore, only the new attribute value together with its timestamp need to be added to the archive. As a result, it would be more space-efficient to use our archiving technique for keeping a relational database that undergoes many changes than to rely on a temporal relational database system for archiving.

9. CONCLUSIONS AND OPEN ISSUES

We have described a novel approach to archiving data based on naturally occurring keys. Perhaps the most contrasting difference with traditional approaches is that traditional diff-based approaches are based on minimizing edit distance and may ignore the historical continuity of elements through time. On the other hand, our technique is "aware" of semantic continuity of elements and therefore, we are able to describe the temporal history of these elements easily. We have shown that such benefit can be achieved without incurring significant space overhead when compared with delta-based repositories in our experiments with real scientific data as well as synthetic data. Moreover, it interacts well with existing compression techniques. For example, the OMIM and Swiss-Prot archives are always within 1% and 8% of the sizes of their respective diff repositories and the compressed archives are always smaller than their respective compressed diff repositories. We have also described how we can extend our archiving tool to an external memory archiver.

We have detailed analysis of the space performance of our archive compared with other approaches. A detailed performance analysis of the external memory archiver, the space efficiency of the proposed indexing schemes as well as actual performance of answering various queries using the proposed indexing schemes, however, have yet been conducted. Another dimension of analysis would be to conduct experiments, for different change ratios and different query workloads, to find out how our archive would perform when compared with delta-based repositories where checkpointing may occur periodically: In the case of our archive, a fresh archive may be created at every kth addition and in the case of a delta-based repository, an entire version of data is stored as a whole for every kth version for some constant k. Our archiver assumes the keys for the data are provided by experts of the database. A natural question is whether the keys can be automatically derived, through data analysis or mining methodologies on various versions. There is also the question on whether we could relax the various requirements we currently impose on keys. Another issue involves changing key structures (or changing schemas); Currently, our archiving technique requires that all versions of the database must conform to the same key structure. Since schemas tend to change slightly over time, a natural question is how this technique can be extended to archive data under circumstances where the key structure may also change.

10. ACKNOWLEDGEMENTS

The authors are grateful to the anonymous reviewers for many useful comments. Peter Buneman is formerly of the University of Pennsylvania and was supported by NSF IIS 99-77408, NSF DL-2 IIS 98-177408 and a Royal Society Wolfson Merit Award. Sanjeev Khanna is supported by an Alfred P. Sloan Fellowship and by NSF Career Award CCR-0093117. Part of this work was done while Keishi Tajima was visiting University of Pennsylvania. Part of this work was done while Wang-Chiew Tan was a Ph.D. student at University of Pennsylvania and was supported by NSF IIS 99-77408 and NSF DL-2 IIS 98-177408.

REFERENCES

Altinel, M. and Franklin, M. J. 2000. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. Cairo, Egypt, 53–64.

Avilla-Campillo, I., Green, T. J., Gupta, A., Onizuka, M., Raven, D., and Suciu, D. 2002. XMLTK: An XML Toolkit for Scalable XML Stream Processing. In *Proceedings of PLAN-X: Programming Language Technologies for XML*. Pittsburg, Pennsylvania.

Bairoch, A. and Apweiler, R. 2000. The SWISS-PROT protein sequence database and its supplement TrEMBL. *Nucleic Acids Research 28*, 45–48.

Buneman, P., Davidson, S., Fan, W., Hara, C., and Tan, W. 2001. Keys for XML. In *Proceedings of the International World Wide Web Conference(WWW10)*. Hong Kong, China, 201–210.

Buneman, P., Deutsch, A., and Tan, W. 1999. A Deterministic Model for Semistructured Data. In *Workshop on Query Processing for Semistructured Data and Non-standard Data Formats*. Jerusalem, Israel, 14–19.

CELLBIODBS. The WWW Virtual Library of Cell Biology. http://vlib.org/Science/Cell_Biology/databases.shtml.

Chawathe, S. S. 1999. Comparing Hierarchical Data in External Memory. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. Edinburg, Scotland, 90–101.

Chawathe, S. S., Abiteboul, S., and Widom, J. 1998. Representing and Querying Changes in Semistructured Data. In *Proceedings of the International Conference on Data Engineering (ICDE)*. Orlando, Florida, 4–13.

Chawathe, S. S. and Garcia-Molina, H. 1997. Meaningful Change Detection in Structured Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Tucson, Arizona, 26–37.

Chawathe, S. S., Rajaraman, A., Garcia-Molina, H., and Widom, J. 1996. Change Detection in Hierarchically Structured Information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Montreal, Canada, 493–504.

Chien, S., Tsotras, V., and Zaniolo, C. 2001. Efficient Management of Multiversion Documents by Object Referencing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. Roma, Italy, 291–300.

Cobena, G., Abiteboul, S., and Marian, A. 2001. Detecting Changes in XML Documents. In *Proceedings of the International Conference on Data Engineering (ICDE)*. Heidelberg, Germany.

CVS. Concurrent Versions System: The open standard for version control. http://www.cvshome.org.

Diao, Y., Fischer, P., Franklin, M. J., and To, R. 2002. Efficient and Scalable Filtering of XML Documents. In *Proceedings of the International Conference on Data Engineering (ICDE)*. San Jose, California.

Driscoll, J. R., Sarnak, N., Sleator, D. D., and Tarjan, R. E. 1989. Making Data Structures Persistent. *Journal of Computer and System Sciences 38,* 1, 86–124.

EMBL-EBI (European Bioinformations Institute). SPTr-XML Documentation. http://www.ebi.ac.uk/swissprot/SP-ML/.

Fontaine, R. L. 2001. A Delta Format for XML: Identifying Changes in XML Files and Representing the Changes in XML. In *XML Europe*. Berlin, West Germany.

Liefke, H. and Suciu, D. 2000. XMill: an Efficient Compressor for XML Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Dallas, Texas, 153–164.

Marian, A., Abiteboul, S., Cobena, G., and Mignet, L. 2001. Change-Centric Management of Versions in an XML Warehouse. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. Roma, Italy, 581–590.

Maruyama, H., Tamura, K., and Uramoto, N. 2000. Digest Values for DOM (DOMHASH). http://www.trl.ibm.com/projects/xml/xss4j/docs/rfc2803.html.

Miller, W. and Myers, E. 1985. A file comparison program. *Software-Practice and Experience 15,* 11, 1025–1040.

Motwani, R. and Raghavan, P. 1995. *Randomized Algorithms*. Cambridge University Press.

Myers, E. 1986. An $O(ND)$ difference algorithm and its variations. *Algorithmica 1,* 2, 251–266.

OMIM 2000. Online Mendelian Inheritance in Man, OMIM (TM). http://www.ncbi.nlm.nih.gov/omim/.

PDOM. GMD-IPSI XQL Engine Version 1.0.2. http://xml.darmstadt.gmd.de/xql/index.html.

PHYSICSCONSTANTS. The NIST Reference on Constants, Units, and Uncertainty. http://physics.nist.gov/cuu/Constants/links.html.

Ramakrishnan, R. and Gehrke, J. 2002. *Database Management Systems*. McGraw-Hill Higher Education, 3rd Edition.

Rochkind, M. 1975. The Source Code Control System. *IEEE Transactions on Software Engineering 1,* 4, 364–370.

Schmidt, A. R., Waas, F., Kersten, M. L., Carey, M. J., Manolescu, I., and Busse, R. 2002. XMark: A Benchmark for XML Data Management. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. Hong Kong, China, 974–985.

Tai, K. C. 1979. The tree-to-tree correction problem. *Journal of the ACM (JACM) 26*, 422–433.

Torp, K., Jensen, C. S., and Snodgrass, R. T. 2000. Effective Timestamping in Databases. *The VLDB Journal 8,* 3-4, 267–288.

Tufte, K. and Maier, D. 2001. Aggregation and Accumulation of XML Data. *IEEE Data Engineering Bulletin 24,* 2, 34–39.

W3C. 1998. Extensible markup language (xml) 1.0. http://www.w3.org/TR/REC-xml.

W3C. 1999a. Namespaces in XML. http://www.w3.org/TR/REC-xml-names.

W3C. 1999b. XML Path Language (XPath). http://www.w3.org/TR/xpath.

W3C. 2000. XML Schema Part 1: Structures. http://www.w3.org/TR/xmlschema-1/.

W3C. 2001a. Canonical XML Version 1.0. http://www.w3.org/TR/xml-c14n.

W3C. 2001b. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/.

XMLTREEDIFF. XML TreeDiff. http://www.alphaworks.ibm.com/formula/xmltreediff.

Zhang, K. and Shasha, D. 1989. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing 18,* 6, 1245–1262.

Zhang, K. and Shasha, D. 1990. Fast algorithms for unit cost editing distance between trees. *Journal of Algorithms 11,* 6, 581–621.

Received September 2002; revised March 2003; accepted;

APPENDIX

A. KEYS FOR XML

Various forms of key specification have been proposed for XML, for example, in XML standard [W3C 1998] and XML Schema [W3C 2000]. We use the system of key specification developed in [Buneman et al. 2001] mainly because it provides a simple method for specifying *relative keys*. Relative keys allow one to define keys in a specified scope and hence one can define a hierarchical set of keys, some of which are relative to others. Here, we briefly review the concept of keys developed in [Buneman et al. 2001].

A.1 XML Model

We model an XML document as a tree whose nodes can be one of the three types: (1) E-node: An E-node is labeled with a tag name, (2) A-node: An A-node is a pair consisting of an attribute name and value, and (3) T-node: A T-node consists of text values only. Only E-nodes can be internal nodes.

A.2 Path Expression

A path expression is a sequence of node names — tag or attribute names. For the purpose of this paper, we consider only the simple fragment of the path language discussed in [Buneman et al. 2001]. The path language consists of the following: (1) the empty path "ϵ", (2) a node name, and (3) the concatenation of paths P/Q where P and Q are paths defined by these rules. We use "/" as the path concatenator just as XPath [W3C 1999b] uses it as a location step separator. In XPath, "/" alone or at the beginning of an XPath expression selects an element above the document root. We disallow concatenations where Q begins with "/".

A.3 Value Equality

The value of a T-node is its data value. The value of an A-node is a pair consisting of its attribute name and its attribute value. The value of an E-node consists of its tag name and two things: (1) a possibly empty list of values of its E and T children nodes according to the document order, and (2) a possibly empty set of values of its A children nodes. Two nodes are *value equal* if they agree on their value, i.e., the trees rooted at the nodes are isomorphic by an isomorphism that is identity on string values according to the type of E-nodes and A-nodes as defined. Finally, we use the notation $n_1 =_v n_2$ to denote the XML value rooted at node n_1 is value equal to the XML value rooted at node n_2.

A.4 Keys

A *key* is a pair $(Q, \{P_1, ..., P_k\})$ where Q and P_i, $i \in [1, k]$ are path expressions. Q is the *target path* and P_i are the *key paths*. Informally, Q identifies a set of nodes – *target set* – reachable from some context node and the target set satisfies the key constraints given by the key paths. This is analogous to relational database where Q is a relation name and P_is form the key for that relation. A formal definition is given below. We denote by $n[\![P]\!]$ the set of nodes reachable from node n via path P.

Definition. A node n *satisfies* a key specification $(Q, \{P_1, \ldots, P_k\})$ if

—For every n' in $n[\![Q]\!]$ and for each $P_i, (1 \leq i \leq k)$, P_i is unique at n'.

—For any n_1, n_2 in $n[\![Q]\!]$, if $n_1[\![P_i]\!] =_v n_2[\![P_i]\!], (1 \leq i \leq k)$, then $n_1 = n_2$.

In the definition above, $=_v$ denotes value equality and $=$ denotes node equality (whether two nodes are the exact same node). This is the definition of *strong* keys given in [Buneman et al. 2001].
Example. The document below does not satisfy the key (`/DB/A`, {`B`}) since both `A` elements have the same key path value, i.e., `<B> 1 </B>`. However, the document satisfies the key (`/DB/A`, {`C`}). Observe that these keys are specified with respect to the root of the document.

```
<DB> <A><B> 1 </B><C> 1 </C></A>   <A><B> 1 </B><C> 2 </C></A> </DB>
```

A.5 Relative Keys

Keys as discussed above are defined with respect to some node. We often would like to describe the key of some node dependent on the key of an ancestor node much like *weak entities* in relational databases [Ramakrishnan and Gehrke 2002]. For example, the key of a weak entity consists of its parent's key and its key (e.g., course Math120, section B). Such dependent keys can be expressed through relative keys as defined below.
Definition. A document satisfies a *relative key* (Q, (Q', S)) if for all nodes n in $[\![Q]\!]$, n satisfies the key (Q', S).
The target path Q' identifies the target set relative to the *context path* Q and Q is always defined with respect to the document root. In other words, "/" is always a prefix of Q. In this paper, we use the word key to mean a relative key.
Example. An example of a set of relative keys can be found in the keys for the company database in Example 2. Versions 1 to 4 of Figure 2 are examples of databases that satisfy these keys.

A.6 Value Inequality

A node n_1 is less than or equal to a node n_2, denoted as $n_1 \leq_v n_2$, if $n_1 <_v n_2$ or $n_1 =_v n_2$. The nodes $n_1 <_v n_2$ if

—n_1 is a T-node, n_2 is not a T-node, or

—n_1 is an A-node, n_2 is an E-node, or

—n_1 and n_2 are both T-nodes and text(n_1) $<$ text(n_2) where text(n) denotes the text string associated with T-node n,

—n_1 and n_2 are both A-nodes and (i) $l_1 < l_2$, or (ii) $l_1 = l_2$ and $v_1 < v_2$ where (l_1, v_1) and (l_2, v_2) are the pairs of attribute name and value associated with nodes n_1 and n_2 respectively, or

—n_1 and n_2 are both E-nodes and (i) tag(n_1) $<$ tag(n_2), or (ii) tag(n_1) $=$ tag(n_2) and children(n_1) $<_l$ children(n_2), or (iii) tag(n_1) $=$ tag(n_2), children(n_1) $=_l$ children(n_2), and attr(n_1) $<_s$ attr(n_2). Next, we define $\leq_l$ and $\leq_s$.
Let children(n_1) be $[c_1, ..., c_m]$ and children(n_2) be $[c'_1, ..., c'_n]$ where each child node is either a T or E-node. Then, $[c_1, ..., c_m] <_l [c'_1, ..., c'_n]$ if
—$m < n$, or
—$m = n$ and $c_i =_v c'_i$ for all $i \in [1, k-1]$, $k \in [1, m]$, and $c_k <_v c'_k$.

We have $[c_1, ..., c_m] =_l [c'_1, ..., c'_n]$ if $m = n$ and $c_i =_v c'_i$ for all $i \in [1, m]$.
Let attr(n_1) and attr(n_2) be $\{l_1 = v_1, ..., l_m = v_m\}$ and $\{l'_1 = v'_1, ..., l'_n = v'_n\}$ respectively where l_is and l'_is are lexicographically ordered. Then,
$\{l_1 = v_1, ..., l_m = v_m\} <_s \{l'_1 = v'_1, ..., l'_n = v'_n\}$ if
—$m < n$, or
—$m = n$, $l_i = l'_i$, $v_i = v'_i$ for $i \in [1, k-1]$, $k \in [1, m]$, $l_k < l'_k$, or
—$m = n$, $l_i = l'_i$, $v_i = v'_i$ for $i \in [1, k-1]$, $k \in [1, m]$, $l_k = l'_k$, and $v_k < v'_k$.
We have $\{l_1 = v_1, ..., l_m = v_m\} =_s \{l'_1 = v'_1, ..., l'_n = v'_n\}$ if $m = n$, $l_i = l'_i$, $v_i = v'_i$ for $i \in [1, m]$.

B. SAMPLE DATA

B.1 Sample record and keys from OMIM

A sample OMIM record is shown below. Some fields of OMIM have been omitted during our translation into XML because they appear in inconsistent formats in older versions of OMIM.

```
<ROOT>
<Record>
   <Num>102579</Num>
   <Title>*102579 REPLICATION FACTOR C, 140-KD SUBUNIT; RFC1</Title>
   <AlternativeTitle>REPLICATION FACTOR C1</AlternativeTitle>
   <AlternativeTitle>RFC*FIELD* TX</AlternativeTitle>
   <Text>Replication factor C is a multisubunit, ... </Text>
   <Contributors>
      <Name>Paul J. Converse</Name>
      <CNtype>updated</CNtype>
      <Date> <Month>11</Month> <Day>16</Day> <Year>2000</Year> </Date>
   </Contributors>
   <Contributors>
      <Name>Jennifer P. Macke</Name>
      <CNtype>updated</CNtype>
      <Date> <Month>8</Month> <Day>27</Day> <Year>1997</Year> </Date>
   </Contributors>
   <Creation_Date>
      <Name>Victor A. McKusick</Name>
      <Date> <Month>12</Month> <Day>14</Day> <Year>1993</Year> </Date>
   </Creation_Date>
</Record>
...
</ROOT>
```

Keys for OMIM data are shown below (many fields in the keys are not reflected in the above sample record):

```
(/, (ROOT, {}))
(/ROOT, (Record, {Num}))
(/ROOT/Record, (Title, {}))
(/ROOT/Record, (AlternativeTitle, {\e}))
(/ROOT/Record, (Text, {}))
(/ROOT/Record, (Ref, {\e}))
(/ROOT/Record, (Contributors,
                {Name, CNtype, Date/Month, Date/Day, Date/Year}))
(/ROOT/Record/Contributors, (Date, {}))
(/ROOT/Record, (Creation_Date, {Name, Date/Month, Date/Day, Date/Year}))
(/ROOT/Record/Creation_Date, (Date, {}))
(/ROOT/Record, (Edit_History, {Name, Date/Month, Date/Day, Date/Year}))
(/ROOT/Record/Edit_History, (Date, {}))
(/ROOT/Record, (Clinical_Synop, {Part, Synop}))
(/ROOT/Record, (See_Also, {Authors, Year}))
(/ROOT/Record, (Allelic_Variants, {Id}))
(/ROOT/Record/Allelic_Variants, (Name, {}))
```

```
(/ROOT/Record/Allelic_Variants, (Text, {}))
(/ROOT/Record/Allelic_Variants, (Mutation, {\e}))
(/ROOT/Record, (Mini_Mim, {\e}))
```

B.2 Sample record and keys from Swiss-Prot

A sample Swiss-Prot record:

```
<ROOT>
<Record>
  <id>100K_RAT</id>
  <class>STANDARD</class> <type>PRT</type> <slen>889</slen>
  <pac>Q62671</pac>
  <mod>
    <date>01-NOV-1997</date><rel>35</rel><comment>Created</comment>
  </mod>
  :
  <protein>
    <name>100 KDA PROTEIN (EC 6.3.2.-).</name>
    <from>Rattus norvegicus (Rat).</from>
    <taxo>Eukaryota</taxo> ...
  </protein>
  <References>
    <Ref>
      <num>1</num><pos>SEQUENCE FROM N.A.</pos>
      <comment>STRAIN=WISTAR</comment>
      <comment>TISSUE=TESTIS</comment>
      <xref> <bib_name>MEDLINE</bib_name><id>92253337</id> </xref>
      <author>Mueller D.</author> <author>Rehbein M.</author> ...
      <title>&quot;Molecular characterization of a novel rat ...</title>
      <in>Nucleic Acids Res. 20:1471-1475(1992)</in>
    </Ref>
    :
  </References>
  <comment>
    <topic>FUNCTION</topic>
    <text>E3 UBIQUITIN-PROTEIN LIGASE WHICH ACCEPTS ... </text>
  </comment>
  :
  <copyright>This SWISS-PROT entry is copyright. ... </copyright>
  <CrossRefs>
    <ref>
      <dbid>EMBL</dbid>
      <primaryid>X64411</primaryid>
      <secid>CAA45756.1</secid>
    </ref>
    :
  </CrossRefs>
  <keywords>
    <word>Ubiquitin conjugation</word><word>Ligase</word>
  </keywords>
  <feature>
    <name>DOMAIN</name>
    <from>77</from><to>88</to><desc>ASP/GLU-RICH (ACIDIC).</desc>
  </feature>
  :
  <sequence>
    <aacid>889</aacid>
    <mweight>100368</mweight>
    <crc><bits>64</bits><checksum>ABD7E3CD53961B78</checksum></crc>
    <seq>
MMSARGDFLN YALSLMRSHN DEHSDVLPVL DVCSLKHVAY VFQALIYWIK AMNQQTTLDT
PQLERKRTRE LLELGIDNED SEHENDDDTS QSATLNDKDD ESLPAETGQN HPFFRRSDSM
TFLGCIPPNP FEVPLAEAIP LADQPHLLQP NARKEDLFGR PSQGLYSSSA GSGKCLVEVT
    :
    </seq>
  </sequence>
</Record>
:
</ROOT>
```

Swiss-Prot keys (many fields in the keys are not reflected in the above sample record):

```
(/, (ROOT, {}))
(/ROOT, (Record, {pac}))
(/ROOT/Record, (sac, {\e}))
(/ROOT/Record, (id, {}))
(/ROOT/Record, (class, {}))
(/ROOT/Record, (type, {}))
(/ROOT/Record, (slen, {}))
(/ROOT/Record, (mod, {date, rel, comment}))
(/ROOT/Record, (protein, {name}))
(/ROOT/Record/protein, (genename, {}))
(/ROOT/Record/protein, (from, {\e}))
(/ROOT/Record/protein, (encoded_on, {\e}))
(/ROOT/Record/protein, (taxo, {\e}))
(/ROOT/Record, (References, {}))
(/ROOT/Record/References, (Ref, {num}))
(/ROOT/Record/References/Ref, (pos, {}))
(/ROOT/Record/References/Ref, (comment, {\e}))
(/ROOT/Record/References/Ref, (xref, {bib_name, id}))
(/ROOT/Record/References/Ref, (author, {\e}))
(/ROOT/Record/References/Ref, (title, {}))
(/ROOT/Record/References/Ref, (in, {}))
(/ROOT/Record, (comment, {\e}))
(/ROOT/Record, (copyright, {}))
(/ROOT/Record, (CrossRefs, {}))
(/ROOT/Record/CrossRefs, (ref, {dbid, primaryid}))
(/ROOT/Record/CrossRefs/ref, (secid, {}))
(/ROOT/Record/CrossRefs/ref, (statusid, {}))
(/ROOT/Record/CrossRefs/ref, (rest, {}))
```

```
(/ROOT/Record, (keywords, {}))
(/ROOT/Record/keywords, (word, {\e}))
(/ROOT/Record, (feature, {name, from, to}))
(/ROOT/Record/feature, (desc, {}))
(/ROOT/Record, (sequence, {}))
(/ROOT/Record/sequence, (aacid, {}))
(/ROOT/Record/sequence, (mweight, {}))
(/ROOT/Record/sequence, (crc, {}))
(/ROOT/Record/sequence/crc, (checksum, {}))
(/ROOT/Record/sequence, (seq, {}))
```

B.3 Sample record and keys from XMark

A sample record from XMark:

```
<site>
  <regions>
    <africa>
      <item id="item1">
        <location>Moldova, Republic Of</location>
        <quantity>1</quantity>
        <name>condemn </name>
        <payment>Money order, Creditcard, Cash</payment>
        <description>
          <text> gold promotions despair flow tempest ... </text>
        </description>
        <shipping>Will ship only within country, Buyer ... </shipping>
        <incategory category="category95"/>
        <incategory category="category48"/>
        :
        <mailbox>
          <mail>
            <from>Javam Suwanda mailto:Suwanda@gmu.edu</from>
            <to>Mehrdad Glew mailto:Glew@cohera.com</to>
            <date>05/28/2001</date>
            <text> back greg flay across sickness ... </text>
          </mail>
          :
        </mailbox>
      </item>
      :
    </africa>
    :
  </regions>
  <open_auction id="open_auction14">
    <initial>132.15</initial>
    <bidder>
      <date>06/23/1998</date> <time>07:37:59</time>
      <personref person="person92"/>
      <increase>9.00</increase>
    </bidder>
    :
    <annotation>
      <author person="person2160"/>
      <description>
        <parlist>
          <listitem> <text> wart varlet metal dark ... </text> </listitem>
          <listitem> <text> modesties marg camp rags ... </text> </listitem>
        </parlist>
      </description>
      <happiness>10</happiness>
    </annotation>
    :
  </open_auction>
  :
</site>
```

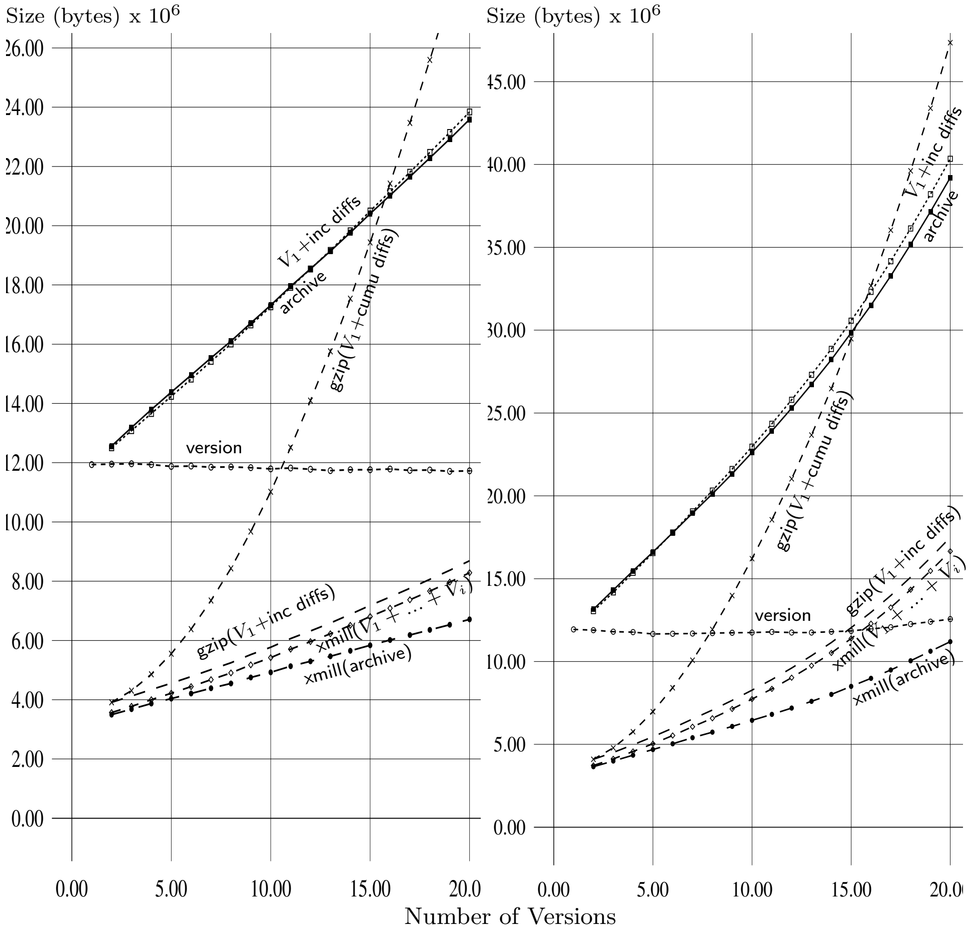
XMark keys are shown below (many fields in the keys are not reflected in the above sample record). In the keys below, we have used "_" for simplicity to denote the values africa, asia, australia, europe, namerica, or samerica.

```
(/, (site, {}))
(/site, (regions, {}))
(/site, (categories, {}))
(/site, (catgraph, {}))
(/site, (people, {}))
(/site, (open_auctions, {}))
(/site, (closed_auctions, {}))
(/site/regions, (africa, {}))
(/site/regions, (asia, {}))
(/site/regions, (australia, {}))
(/site/regions, (europe, {}))
(/site/regions, (namerica, {}))
(/site/regions, (samerica, {}))
(/site/regions/_, (item, {id}))
(/site/regions/_/item, (featured, {\e}))
(/site/regions/_/item, (location, {}))
(/site/regions/_/item, (quantity, {}))
(/site/regions/_/item, (name, {}))
(/site/regions/_/item, (payment, {}))
(/site/regions/_/item, (description, {}))
(/site/regions/_/item, (shipping, {}))
(/site/regions/_/item, (incategory, {category}))
(/site/regions/_/item, (mailbox, {}))
(/site/regions/_/item/mailbox, (mail, {from, to, date, text}))
(/site/categories, (category, {id}))
(/site/categories/category, (name, {}))
(/site/categories/category, (description, {\e}))
(/site/catgraph, (edge, {from, to}))
(/site/people, (person, {id}))
(/site/people/person, (name, {}))
(/site/people/person, (emailaddress, {\e}))
```

```
(/site/people/person, (phone, {\e}))
(/site/people/person, (address, {\e}))
(/site/people/person, (homepage, {\e}))
(/site/people/person, (creditcard, {\e}))
(/site/people/person, (profile, {\e}))
(/site/people/person, (watches, {\e}))
(/site/open_auctions, (open_auction, {id}))
(/site/open_auctions/open_auction, (initial, {}))
(/site/open_auctions/open_auction, (reserve, {\e}))
(/site/open_auctions/open_auction,
 (bidder, {date, time, personref/person, increase}))
(/site/open_auctions/open_auction/bidder, (personref, {}))
(/site/open_auctions/open_auction, (current, {}))
(/site/open_auctions/open_auction, (privacy, {\e}))
(/site/open_auctions/open_auction, (itemref, {}))
(/site/open_auctions/open_auction/itemref, (item, {}))
(/site/open_auctions/open_auction, (seller, {}))
(/site/open_auctions/open_auction/seller, (person, {}))
(/site/open_auctions/open_auction, (annotation, {}))
(/site/open_auctions/open_auction/annotation, (author, {}))
(/site/open_auctions/open_auction/annotation/author, (person, {}))
(/site/open_auctions/open_auction/annotation, (description, {}))
(/site/open_auctions/open_auction/annotation, (happiness, {}))
(/site/open_auctions/open_auction, (quantity, {}))
(/site/open_auctions/open_auction, (type, {}))
(/site/open_auctions/open_auction, (interval, {start,end}))
(/site/closed_auctions,
 (closed_auction, {seller, buyer, itemref/item, date}))
(/site/closed_auctions/closed_auction, (itemref, {}))
(/site/closed_auctions/closed_auction, (price, {}))
(/site/closed_auctions/closed_auction, (annotation, {}))
(/site/closed_auctions/closed_auction/annotation, (author, {}))
(/site/closed_auctions/closed_auction/annotation/author, (person, {}))
(/site/closed_auctions/closed_auction/annotation, (description, {}))
(/site/closed_auctions/closed_auction/annotation, (happiness, {}))
(/site/closed_auctions/closed_auction, (quantity, {}))
(/site/closed_auctions/closed_auction, (type, {}))
```

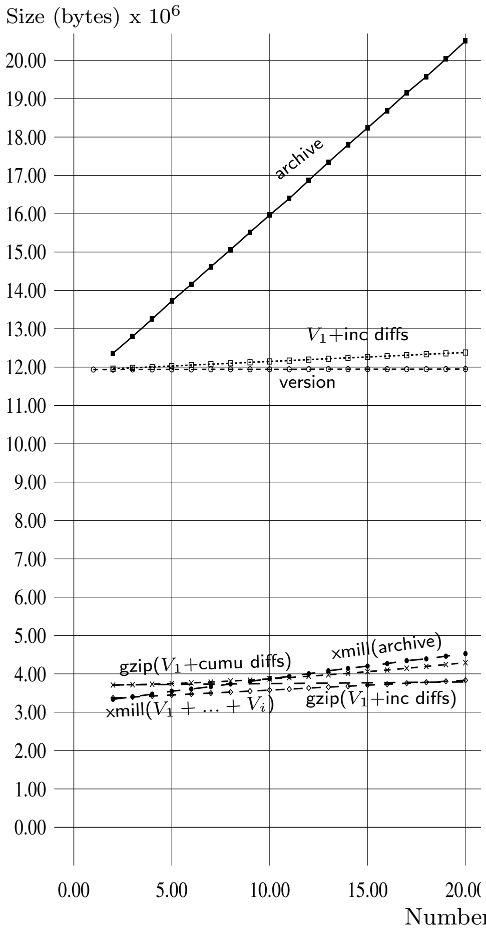
C. ADDITIONAL EXPERIMENTAL RESULTS

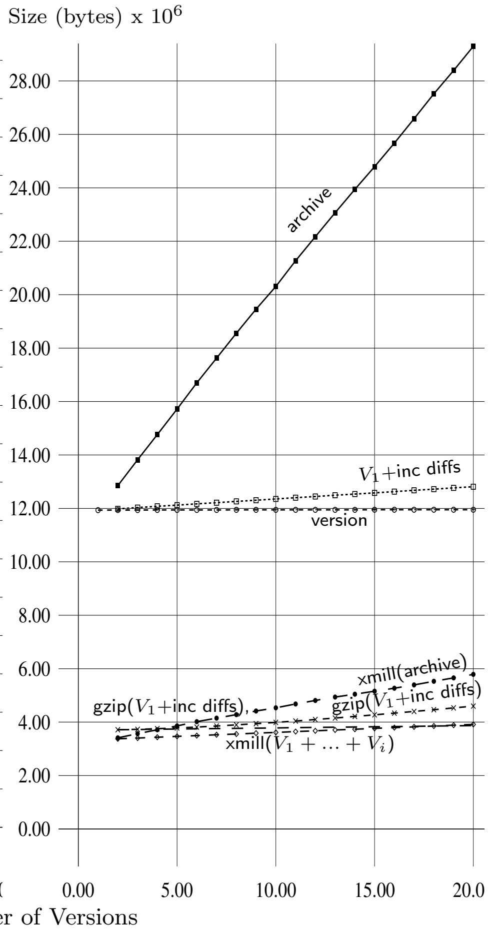
C.1 Storage performance on XMark under different change ratio



(a) 3.33%/3.33%/3.33% change ratio (b) 6.66%/6.66%/6.66% change ratio

C.2 Storage performance on XMark with insertion and deletion of highly similar elements





(a) 3.33%/3.33%/3.33% change ratio (b) 6.66%/6.66%/6.66% change ratio